

Processing Graph Method - GUI Handbook

[License](#)

[Glossary](#)

[Introduction](#)

- [Introduction to the GUI](#)
- [How to create a new graph specification?](#)

[Working with Graph State Files](#)

- [Creating a new GSF](#)
- [Opening a GSF](#)
- [Closing the current GSF](#)
- [Saving the current GUI graph](#)
- [Saving the current GUI graph in a GSF in a user-specified directory](#)
- [Opening a recent GSF](#)
- [Exiting the GUI](#)

[Forms - general notes](#)

- [Buttons OK, Cancel, Apply, Validate Form, Print](#)
- [Non-editable fields](#)

[and forms](#)

- [Tables](#)
- [Family Tree tables](#)
- [Type names](#)
- [Nested strings](#)
- [Printing forms](#)

Icon and Arc Forms

- [Icon Prototype Form](#)
- [Icon Call Form](#)
- [Icon Arc Form](#)
- [Arc Form](#)

Exterior Forms

- [Prototype Form](#)
- [Port Association Form](#)
- [Banner Form](#)
- [Type List Form](#)
- [Included Graph List Form](#)

Editing

- [Editing forms](#)
- [Editing graphic elements](#)

Validation

- [Validating forms](#)
- [Validating the graph](#)

Translation

- [Translation](#)

Miscellaneous

- [Batch execution](#)

- [Support for C++](#)
- [Format of GSF files](#)

Example

- [Exterior forms](#)
- [Icon prototype forms](#)
- [Call forms](#)
- [Icon arc forms](#)
- [Arc forms](#)
- [System supplied prototype forms](#)

References

Processing Graph Method - GUI Handbook

by

Michal Iglewski

July 17, 2002

The Processing Graph Method Tool (PGMT) product is being released under the GNU General Public License Version 2, June 1991 and related documentation under the GNU Free Documentation License Version 1.1, March 2000.

<http://www.gnu.org/licenses/gpl.html>

Glossary

Expression

An expression according to the rules of the programming language supported by PGM, i.e., C++. Operands may be literal numbers, variable names, indexed arrays, and function calls.

Leaf

A family with height being 0.

Full path

The full path of a file. In a Unix file system, the directory separator is a slash '/'.

NestedString

An aggregate used to denote a family.

Parsing tree

GUI memory, i.e., the internal data structures of the GUI graph.

Type name

A type name is a name of either a language-defined type or a user-defined type. The language currently supported by PGM is C++. In C++, language-defined type names differ from variable names and can be composed of several names like *unsigned int*, *long double*, etc.

A user-defined type name is a single name. The corresponding type is defined in one of the files in the [Type List](#).

User-defined data type (or user-defined class)

A data type (or class) defined in the target language (C++). A PGM user can define a base type for a PGM family by creating a C++ class and in the case of templated classes, by providing a type definition (typedef) for each distinct instantiation of the templated class.

Variable name

Name used to denote one of the following entities: prototype, icon family, port family, index, formal type argument, formal mode argument, formal GIP, user-defined type.

The first character is alpha, and each subsequent character is either alphanumeric or an underscore. No embedded blanks are allowed.

Introduction

Introduction to the PGMT/GUI

This document describes the Graphic User Interface (GUI) of the Processing Graph Method Tool ([PGMT](#)).

In the following, we refer to the human using the GUI as the *user*, and a *GUI session* refers to the editing of a GUI graph.

While the user is editing the GUI graph, the internal data structures in the GUI memory are called the *parse tree*. The user may elect to save the GUI graph in a file, called the *Graph State File*, or *GSF*. After saving the GUI graph in a GSF, the user may start a new GUI session and open the previously saved GSF. The state of the parse tree and the information presented to the user are identical to what they were when the GUI graph was saved.

How to create a new graph specification

Building the graph consists of placing and connecting the icons of the data flow graph on the screen and filling out forms describing the icons and their interconnections. The entire process can be done in one or more sessions.

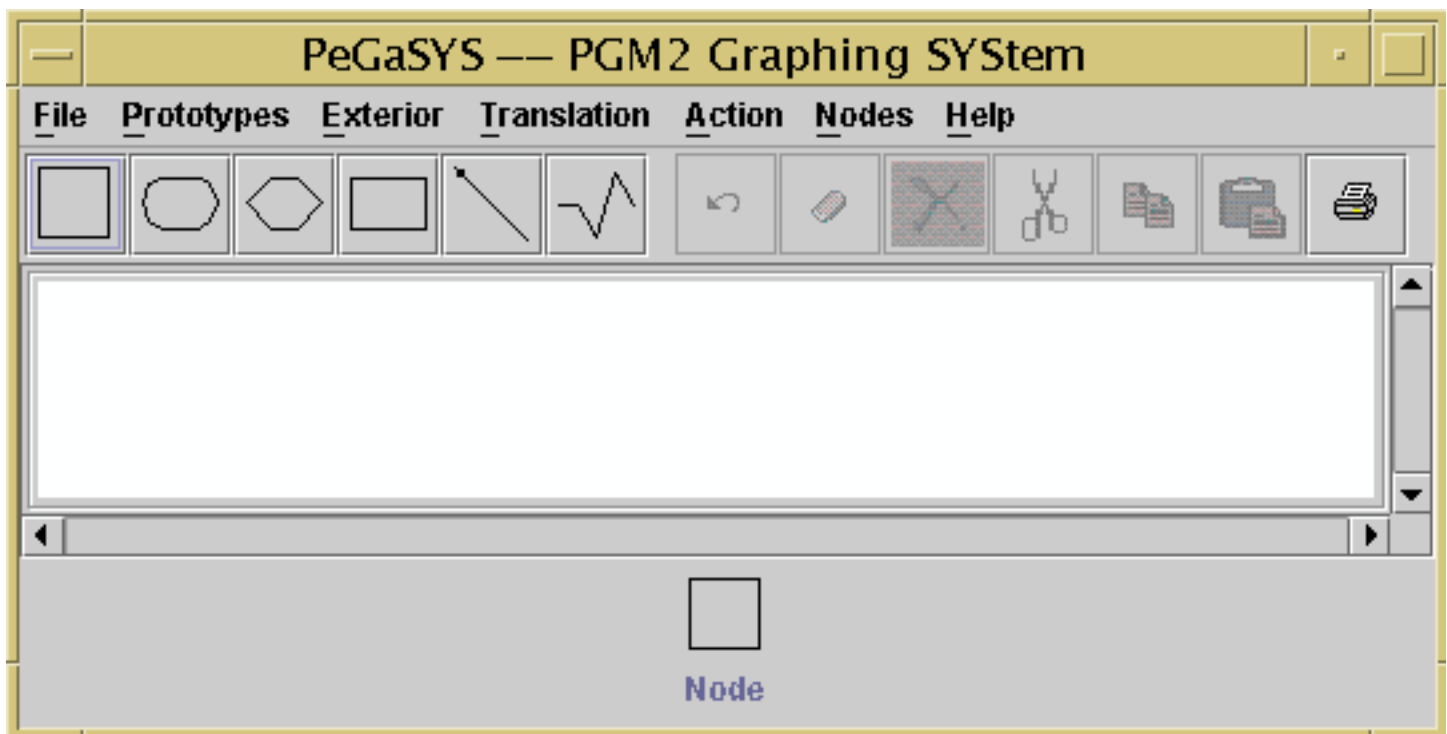
The specific actions are usually executed in the following order:

1. [creating](#) a new Graph or [opening](#) an existing Graph State File,
2. filling out the [Banner Form](#),
3. [filling](#) out the list of user-defined data types used in the Graph,
4. [filling](#) out the list of graph prototypes used in the Graph,
5. filling out [Prototype Form](#) for the graph,
6. defining User-specific Node [Prototypes](#),
7. [placing Icons](#) on the Screen,
8. filling out [Icon Forms](#),
9. [connecting](#) Icons with Arcs,
10. filling out [Arc Forms](#),
11. [connecting](#) the Graph Exterior to Ports in the Graph,
12. [validating](#) the Graph Specification,
13. [generating](#) C++ code,
14. [exiting](#) the GUI.







Anytime during this process, the user can [save](#) the Graph.

Placing Icons on the Graph

The user places the icons (transitions, places, and/or included graphs) in the graph screen window. The user first selects from the tool the type of icon that is to be placed on the screen by clicking with the left mouse button the corresponding icon on the tool bar, or if the tool bar is hidden, by selecting the corresponding icon from the Nodes Menu.



The meaning of icons is defined by the following table:

Icon	Meaning
	Select icon
	Transition icon
	Place icon
	Included Graph icon
	Arc icon
	Arc bends icon

The arrow icon changes into a small dark icon whose shape is the same as the icon to be placed on the screen. The user then moves the icon to the desired place on the screen and left clicks to place it on the screen. This procedure is repeated until all of the icons for the graph are placed on the screen. If a user wishes to delete an icon, the select icon (the left most button on the toolbar) is chosen and the user left clicks on the icon to be removed (the icon will turn red when it is selected). The icon is removed by clicking on the delete button (scissors) on the tool bar. Once all of the tokens have been

placed on the screen the user should return to select mode by left clicking on the toolbar select button.

For more information about placing and connecting the icons on the screen, go to [Editing Graphic Elements](#), and for more information about filling out forms, go to [Editing Forms](#).

Working with Graph State Files

Start a new session

To start a new session, select NEW from the Files menu. The GUI graph is initially blank.

Create a new GSF

To create a new GSF, select NEW from the Files menu. This opens a new empty graph with the Graph Class Name being New (or New*n* if the files New.gsf, New0.gsf, New1.gsf,..., New*n-1*.gsf already exist in the current directory).

Open a GSF

To open an existing GSF, select OPEN from the Files menu.

When opening an existing GSF, the GUI verifies that the stem of the file name is the same as the Prototype Name in the Prototype Form. If they are different, the GUI informs the user and suggests to change the Prototype Name to the stem of the file name.

The file name in the banner is always set to the name of the opened file.

Only one GSF can be opened at time. If any changes have been made to the GUI graph since it was last saved, the GUI asks the user whether to save it before opening a new file.

Close the current GSF

To close the current GSF, close all editable forms and select CLOSE from the Files menu.

If any changes have been made to the GUI graph since it was last saved, the GUI asks the user whether to save it before closing.

Save the current GUI Graph

To save the current GUI Graph in a GSF, close all editable forms and select SAVE from the Files menu.

The file name of the GSF will be the Graph Class Name (specified by the user in the Prototype Name in the Graph Prototype Form) appended with extension ".gsf".

The GUI Graph is saved in a GSF in the format specified by the [GSF Specification](#).

Save the current GUI Graph in a GSF in a user-specified directory

To save the current GUI Graph in a GSF in a specific directory, close all editable forms and select SAVE AS from the Files menu.

The file name of the GSF will be the Graph Class Name (specified by the user in the Prototype Name in the Graph Prototype Form) appended with extension ".gsf".

Open a recent GSF

To reopen a recently edited GSF, close all editable forms and select the file from the Recent Files sub-menu of Files menu. The sub-menu contains the list of files edited in this session.

If any changes have been made to the GUI graph since it was last saved, the GUI asks the user whether to save it before closing.

Exit the GUI

To end the current session, close all editable forms and select EXIT from the Files menu.

If any changes have been made to the GUI graph since it was last saved, the GUI asks the user whether to save it before exiting.

Forms - General Notes

All forms associated with the graph are opened by selection from the Exterior Menu and they are discussed in [Exterior Forms](#) section. The forms for icons and arcs are opened by pointing at an icon or and an arc, and pressing the right mouse button. They are discussed in [Icon and Arc Forms](#) section.

Buttons OK, Cancel, Apply, Validate Form, Print

Every form has the buttons *OK* and *Print*, and possibly some of the following buttons: *Apply*, *Cancel*, *Validate Form*, *Open Body*, *Read File*. The *Print* button is used to send the form to a printer, or to print it to a file. The meaning of other buttons depends on the kind of the form and on whether the form is [editable](#) or not.

If the form is editable, then it usually contains the buttons *Apply* and *Cancel*. When the user clicks on one of these buttons, the following respective actions occur:

- *Apply*: Perform the syntax check of every field in the form and report all errors to the user. The form does not have to be completed; blank fields are acceptable. If there are no errors, update the [parse tree](#) in memory to incorporate changes made in the form.
- *OK*: Perform the actions described for *Apply* and then close the form.
- *Cancel*: Close the form without checking against the syntax rules and without making any changes to the parse tree in memory.

If the form is non-editable, clicking on the *OK* button closes the form.

Two of the following forms, Family Tree table, Transition Statement, may be editable and do not contain the *OK* and *Apply* buttons. To incorporate the changes made to such a form, the user should close it, and click either the *OK* or *Apply* button on the respective mother form.

Some of the editable forms contain the *Validate Form* button. If the user clicks on this button, the GUI perform the local semantics check of every field in the form and report all errors to the user. Note that the full semantics checks are made when the user selects "Validate Graph" in the Translation Menu.

The *Open Body* button is used by Transition Prototype Forms and the *Read File* button is only used by the [Transition Statement Form](#).

Non-editable fields and forms

In many situations, a field contains the information that is obtained from other fields in the same form or from other forms. In such situations, the field itself cannot be editable and to attract the user's attention, the field background is of different, yellow/orange color.

Example:

PGMT											
Call Form											
Icon Family Name: <input type="text" value="wildOne"/>											
Prototype Name: <input type="text" value="PassOn"/>											
Icon Family Tree											
<input type="button" value="add"/>	<table border="1"> <thead> <tr> <th>Index</th> <th>Lower bound</th> <th>Upper bound</th> </tr> </thead> <tbody> <tr> <td>k</td> <td>1</td> <td>breadth</td> </tr> </tbody> </table>	Index	Lower bound	Upper bound	k	1	breadth				
Index	Lower bound	Upper bound									
k	1	breadth									
<input type="button" value="del"/>											
Actual Type Arguments											
<table border="1"> <thead> <tr> <th>Formal Type</th> <th>Base Type</th> </tr> </thead> <tbody> <tr> <td>T</td> <td>char</td> </tr> </tbody> </table>		Formal Type	Base Type	T	char						
Formal Type	Base Type										
T	char										
Actual Mode Arguments											
<table border="1"> <thead> <tr> <th>Formal Height</th> <th>Actual Height</th> <th>Formal Base Type</th> <th>Actual Base Type</th> </tr> </thead> <tbody> <tr> <td></td> <td></td> <td></td> <td></td> </tr> </tbody> </table>	Formal Height	Actual Height	Formal Base Type	Actual Base Type							
Formal Height	Actual Height	Formal Base Type	Actual Base Type								
Actual GIP Bindings											
<table border="1"> <thead> <tr> <th>Formal GIP</th> <th>Family Tree</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>how_wide</td> <td><input type="checkbox"/> Fix Family Tree</td> <td>2 * width</td> </tr> <tr> <td>how_long</td> <td><input type="checkbox"/> Fix Family Tree</td> <td>length</td> </tr> </tbody> </table>	Formal GIP	Family Tree	Value	how_wide	<input type="checkbox"/> Fix Family Tree	2 * width	how_long	<input type="checkbox"/> Fix Family Tree	length		
Formal GIP	Family Tree	Value									
how_wide	<input type="checkbox"/> Fix Family Tree	2 * width									
how_long	<input type="checkbox"/> Fix Family Tree	length									
Initial Value											
Queue Token Index:	<input type="text"/>	LowerBound:	<input type="text"/>								
		UpperBound:	<input type="text"/>								
<table border="1"> <thead> <tr> <th>Index</th> <th>Lower bound</th> <th>Upper bound</th> </tr> </thead> <tbody> <tr> <td></td> <td></td> <td></td> </tr> </tbody> </table>	Index	Lower bound	Upper bound								
Index	Lower bound	Upper bound									
Value: <input type="text"/>											
<input type="button" value="OK"/> <input type="button" value="Cancel"/> <input type="button" value="Apply"/> <input type="button" value="Validate Form"/> <input type="button" value="Print"/>											

The user cannot edit two different forms at the same time. If one of the forms is being already edited, the user will not be allowed to open another form for editing purposes. However, in many situations, the user will be allowed to consult one or more forms at the same time. All fields in the forms being consulted will be of different yellow/orange color. There is one exception to this rule. If the form being consulted contains a field with a button to open another form, the field background is of white color but all fields in the associated form will be of yellow/orange color.

Example:

Output Ports				
Family Name	Category	Token Ht	Base Type	Family Tree
passOut	transition	2	T	<input type="checkbox"/> Exp Family ...
feedBackOut	transition	0	unsigned int	<input type="checkbox"/> Exp Family ...

Tables

Filling out some of the forms requires the construction of tables where the number of rows in the table is application dependent. A table with a fixed number of rows is called a *fixed table*. A table with variable number of rows is called an *expandable table*.

The addition and deletion of rows to a table are performed by the use of the *add* and *del* buttons located on the left hand side of the table.

To select a field in the table, the user moves the cursor over an entry of the table and then left clicks. Initially tables to be constructed are empty. To add a new row, the user selects a field and then clicks the *add* button.

- If a field in the table is selected and the user clicks the *add* button, the new row will follow the row containing the selected field. All existing rows below the inserted row will be moved down.
- If no field in the table is selected and the user clicks the *add* button, the new row will be at the top of the table, and all existing rows will be moved down.

To delete a row, the user selects a cell and then clicks the *del* button. The row containing the selected field will be removed.

Clicking on the table header clears the current selection of fields. This can be useful if the user wants to add a new row at the top of the table, and one of its fields is already selected.

Family Tree tables

Family Tree table is a description of a family.

Example:

Index	Lower Bound	Upper Bound
r	0	how_long - 1
s	0	how_wide - 1

Each Family Tree table satisfies the following conditions:

- each Index is a variable name unique within the Family Tree,
- each Lower and Upper Bound is an expression,
- each operand in the Lower and Upper Bound expressions is a literal, a GIP, or a previously occurring index in the Family Tree, i.e., an index in a row above the expression.

In each row, the Lower Bound and Upper Bound expressions, when evaluated, give the inclusive limiting values of the index. Intuitively, one can imagine that a family tree specifies a nested loop, where the first row gives the loop variable and bounds of the outermost loop.

Not every Family Tree table is expandable. If a Family Tree table is fixed, the buttons to add and delete are omitted.

It is possible for a Family Tree to occupy a cell in a table. If so, then every cell in the column is occupied by a Family Tree, and all are expandable or all are fixed. An expandable Family Tree is represented by a square button followed by "Exp Fam Tree" ([example](#)), and a fixed Family Tree is represented by a square button followed by "Fix Fam Tree" ([example](#)).

If the user double-clicks the button in one of the cells, a Family Tree Form opens for the user to edit. To incorporate the changes made to the form, the user should close it, and click either the *OK* or *Apply* button on the respective mother form.

Type Names

A Type Name is a name of either a language-defined type or a user-defined type. The language currently supported by PGMT is C++. In C++, language-defined type names differ from variable names and can be composed of several names like *unsigned int*, *long double*, etc.

A user-defined type name is a single name. The corresponding type is defined in one of the files in the [Type List](#).

NestedStrings

A NestedString denotes a family; all elements of a properly formed NestedString must have the same height.

A NestedString is a list of elements separated by commas and delimited by braces { }. The elements in the list can be either expressions or (recursively) NestedStrings. The complete syntax is presented in [Format of GSF files](#) section.

Example:

{ 1, 5, 3, 2 } is a NestedString of height 1.

{ { 1, 5, 3, 2 }, { 17 }, {}, { 4, 2 } } is a NestedString of height 2.

Note that all non-null NestedString elements of a NestedString must have the same level of nested braces.

A null NestedString of any height is denoted by {}. It implies that for some NestedStrings we can only determine the minimal height.

Example:

{ { {} }, {} } is a NestedString whose height must be at least 3, but can be any height greater than 3.

The actual height must be determined from the token height of the place in which the NestedString specifies the initial value.

- If the place is a graph variable, the height must be the same.
- If the place is a queue, then the height must be one greater, to allow for multiple tokens, each having the specified token height.

Printing forms

Each of the forms contains the Print button. Clicking it will open a dialog asking the user to specify the scaling. Without scaling down the printed form, it would be greater than the original form and might be greater than one page. To obtain the resolution similar to that on the screen, the form should be reduced to about 80%. The disposition of objects on the printed form might slightly differ from that on the original form.

The reduced form is previewed on the screen before the user makes the final decision about printing it.

Icon and Arc Forms

In this section we describe how to open the various forms for each icon and for arcs. There are three kinds of icons: transition icons, place icons, and included graph icons. For each icon, there are three kinds of forms, each of which may be opened by pointing at the icon and pressing the right mouse button. This opens a pop-up menu from which the user can select one of three forms associated with the icon. The three forms are the Prototype Form, the Call Form, and the Icon Arc Form. Finally, there is an Arc Form for each arc, which specifies how the ports of two respective nodes should be connected.

Icon Prototype Form

Every icon has a Prototype Form. It is possible for several icons to have the same Prototype Form. The Prototype Form for a given icon may be opened read-only from the icon pop-up menu. Some Prototype Forms (i.e., for Pack and Unpack Transitions and for standard Queues and Graph Variables) are read-only, and the operator cannot edit them.

To open and edit an Ordinary Transition Prototype Form or non-standard Place Prototype Form, the operator selects the desired Prototype Form from the Prototype Menu on the menu bar. To edit the Prototype Form for an Included Graph, the operator uses the GUI to open the GSF of the underlying graph and edits the Graph Prototype Form for the underlying graph.

The structure of Prototype Forms is explained in the [Exterior Form](#) section.

Icon Call Form

Every icon has a unique Call Form. The Call Form identifies the Prototype Form for that icon and specifies the bindings of the formal arguments specified in the Prototype Form.

To open and edit the Call Form for a given icon, the operator points at the icon, presses the right mouse button, and selects the Call Form in the pop-up menu. Before an icon's Call Form can be opened, the Prototype Form for the icon must be identified. If the icon's Prototype Form has not been identified, the GUI opens a pop-up menu with the existing prototype names in the graph, and asks the user to select the desired prototype name.

- If the icon is a Place Icon, the Prototype Name has to be "[Queue](#)", "[GVar](#)" or the Prototype Name of an user-defined non-standard queue or graph variable.
- If the icon is a Transition Icon, the Prototype Name has to be "[Pack](#)", "[Unpack](#)", or the Prototype Name of an user-defined ordinary transition.
- If the icon is an Included Graph Icon, the Prototype Name has to be one of the Prototype Names in the [Included Graph List](#).

The form below depicts a general Call Form, which covers all kinds of icons. If the prototype has type arguments, mode arguments and/or associated GIP's, then the form will contain white boxes where the actual values of these items must be entered.

PGMT				
Call Form				
Icon Family Name:		<input type="text" value="label1"/>		
Prototype Name:		<input type="text" value="NewProt"/>		
Icon Family Tree				
<input type="button" value="add"/> <input type="button" value="del"/>	Index	Lower bound	Upper bound	
	<div></div>			
Actual Type Arguments				
Formal Type		Base Type		
<div></div>				
Actual Mode Arguments				
Formal Height	Actual Height	Formal Base Type	Actual Base Type	
<div></div>				
Actual GIP Bindings				
Formal GIP	Family Tree	Value		
<div></div>				
Initial Value				
Queue Token Index:		<input type="text" value=""/>	LowerBound:	<input type="text" value=""/>
			UpperBound:	<input type="text" value=""/>
Index	Lower bound	Upper bound		
<div></div>				
Value:		<input type="text" value=""/>		
<input type="button" value="OK"/>	<input type="button" value="Cancel"/>	<input type="button" value="Apply"/>	<input type="button" value="Validate Form"/>	<input type="button" value="Print"/>

Icon Family Name

The GUI provides a default name of the icon that will be usually changed by the user. The Icon Family Name has to be unique among Icon Family Names, Prototype Names and Formal Names in the GUI Graph Prototype Form.

Prototype Name

The Prototype Name is set by the GUI according to the rules described at the beginning of this section.

Icon Family Tree

A single icon may represent a family of icons. If the icon does represent a family, then a description of the family is provided in a [Family Tree table](#) labeled Icon Family Tree. The number of entry lines in the family table is the same as the height of the family and the index variable and upper and lower bounds for each level of the tree must be specified.

Each operand in the Lower and Upper Bound Expressions is a literal number, a Formal GIP with height = 0 in the GUI Graph Prototype Form, or an index in a row of the Icon Family Tree above the expression.

Actual Type Arguments

There is one row for each Formal Type Argument in the Icon's Prototype Form. In the left column, the GUI lists the Formal Type Arguments in the Prototype Form of the Icon. In the right column, the user enters a [Type Name](#) in each row.

Actual Mode Arguments

There is one row for each Formal Mode Argument in the Prototype Form. In the Formal Height and Formal Base Types columns, the GUI lists the respective Formal Height and Formal Base Type in the Formal Mode Arguments of the Icon's Prototype Form. In each Actual Height column, the user enters a non-negative integer. In each Actual Base Type, the user enters a [Type Name](#).

Actual GIP Bindings

There is one row for each Formal GIP in the Prototype Form.

In the Formal GIPs column, the GUI lists the respective Formal GIPs in the icon's Prototype Form.

Each element in the second column is a [Fixed Family Tree](#) with the number of rows equal to the height of the Formal GIP. Each operand in the Lower and Upper Bound expressions is a non-negative literal number, a Formal GIP with height = 0 in the GUI Graph, an index in the Icon Family Tree, or an index in a row above the expression in the same Family Tree.

The Value Field is either an expression or a [NestedString](#).

- If the user enters a NestedString in the Value column, then the respective Family Tree is ignored. Each operand in the expressions in the NestedString is a literal of the Base Type specified for the respective Formal GIP in the Icon's Prototype Form, a Formal GIP with height = 0 in the GUI Graph Prototype Form, or an index in the Icon Family Tree.
- The user may enter a value that is the name of a Formal GIP of the GUI Graph, provided its height and base type respectively match the height and base type of the Formal GIP of the Icon's Prototype. In this case the respective Family Tree will be ignored.

Note: This is not supported in the current version of the translator.

- Except as described in the two immediately preceding bullets, the user enters an expression in the Value column. Each operand in the expression is a literal number, a Formal GIP with height = 0 in the GUI Graph Prototype Form, an index in the Icon Family Tree, or an index in the respective Family Tree.

Initial Value

The area below the heading "Initial Value" is used for initializing a place.

- If the icon is a transition or included graph then this area of the screen is inactive (as transitions and included graph do not have values).
- For a Queue the area may be filled with one or more tokens that will be used to initialize the queue when the graph is constructed.
- For a Graph Variable this area is filled with exactly one token that initializes the graph variable, and so the Token Index, together with its Lower and Upper Bounds are inactive.

The Lower Bound of the Queue Token Index (if applicable) is "1", read-only.

The Queue Token Index (if applicable) and the fixed table below it comprise a Family Tree. Below the Queue Token Index, the number of rows in the Family Tree table is equal to the Actual Height specified in the Actual Mode Argument. Note: the Prototype Form for a place has exactly one Formal Mode Argument. Thus there is exactly one Actual Height, which specifies the height of the initial token(s) in the place.

The Value Field is either an expression or a NestedString.

- If the Entity Type in the Prototype Form is "gvar", then the user supplies an initial value (Observe that in this case, the Queue Token Index is inactive).
- If the Entity Type is "queue", then the user has the option of supplying an initial value. The Queue Token Index identifies each of possibly many tokens. The Upper Bound specifies how many tokens, and is 0, by default, to indicate that the Queue is initially empty. In this case, all other fields under the heading "Initial Value" are left blank.

The following rules apply for a queue with at least one initial token and for a graph variable, which must be initialized with exactly one token.

- Each operand in the Lower and Upper Bound expressions is a literal number, a Formal GIP with height = 0 in the GUI Graph, an index in the Icon Family Tree, or an index in a row above the expression.
- If the user enters a NestedString in the Value field, then the Family Tree, including the Queue Token index (if applicable), will be ignored.
 - Each operand in the expressions in the NestedString is a literal number, a Formal GIP with height = 0 in the GUI Graph Prototype Form, or an index in the Icon Family Tree.
 - If the Entity Type is "queue", then the level of nesting in the NestedString is equal to one more than the Actual Token Height in the Actual Mode Argument.
 - If the Entity Type is "gvar", then the level of nesting in the NestedString is equal to the Actual Token Height in the Actual Mode Argument.
- The user may enter a Value that is the name of a Formal GIP of the GUI Graph, provided its height and base type respectively match the height and base type of the Actual Mode Argument of the Place specified above in the Call Form. In this case the Family Tree will be ignored. Assuming that the GIP is intended to provide the value of each

token, the Queue Token Index (if applicable) will be retained.

Note: This is not supported in the current version of the translator.

- Except as noted in the two immediately preceding bullets, the user enters an expression in the Value field. Then each operand in this expression is a literal number, a Formal GIP with height = 0 in the Graph Prototype Form, an index in the Icon Family Tree, or an index in the Initial Value Family Tree.

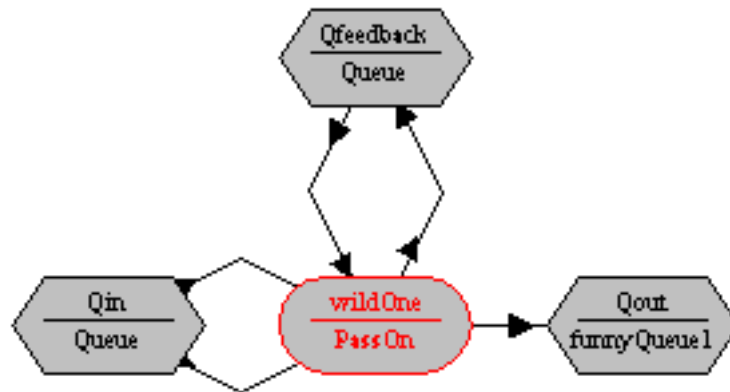
Icon Arc Form

The Icon Arc Form for a given icon displays some of the arc information for every arc connected to the selected Icon. The information includes

- the icon's Port Family Name,
- the Icon Family Name of the other icon (i.e., at the other end of the arc),
- the Port Family Name of the other Icon.

To open the Icon Arc Form for a given icon, the operator points at the icon, presses the right mouse button, and selects the Icon Arc Form in the pop-up menu. The Icon Arc Form is read-only.

Example: clicking at the transition icon, wildOne, in the example [ex1.gsf](#)



will open the following Icon Arc Form:

PGMT

Icon Arc Form

Icon Family Name:

Input Ports

Input Port Family Na...	From Icon Family Na...	From Port Family Name	Open Arc Form
feedBackIn	Qfeedback	OUTPUT	<input type="checkbox"/> open ...
passIn	Qin	OUTPUT	<input type="checkbox"/> open ...
passIn	Qin	OUTPUT	<input type="checkbox"/> open ...

Output Ports

Output Port Family N...	To Icon Family Name	To Port Family Name	Open Arc Form
passOut	Qout	INPUT	<input type="checkbox"/> open ...
feedBackOut	Qfeedback	INPUT	<input type="checkbox"/> open ...

To view the full Arc Form read-only for any of the connected arcs, the user clicks the respective button in the fourth column.

Arc Form

The arc form is used to specify connections between families of ports of two icons.

To edit an arc form for a given arc, the user points at the arc and click the right mouse button. This identifies one icon at each end of the arc on the screen. The icon at the tail end of the arc is the *From Icon*, and the icon at the head end of the arc is the *To Icon*. Filling out the Arc Form is possible if the prototypes of two icons have been defined. If this is not the case, the user will get an error message.

Before the Arc Form of an arc can be opened, the Port Family Names of both icons must be identified. This requires that the respective Icon's Call Form and Prototype Form have been created and filled out. The reason for this requirement is that the information in these forms is needed to display the correct table sizes and family indices for each Port Family. If these forms have not been filled out, or the GUI cannot find two eligibles port names, the GUI produces an error message.

An output port of the From Icon and an input port of the To Icon are *eligible to be connected* if

- the modes (i.e., the Token Height and Base Type) of the two ports (specified in the respective Prototype Forms of the icons) are the same, and
- the categories of the two Ports (specified in the respective Prototype Forms of the icons) are different (i.e., one port has Category "place", and the other is "transition").

If the Port Family Names have not been identified yet, and there are two or more output ports that may be eligible, or two or more input ports that may be eligible, the GUI opens a pop-up menu with these port names, and asks the user to select

the desired Port Family Name. If both Port Family Names have been identified, the Arc Form is opened.

Note that the user may specify an expression for the Token Ht in the Prototype Form and, in this case, the GUI has no way to check that the Token Height of the two ports are the same, and that these ports are eligible to be connected.

Example:

PGMT

Arc Form

Nested Loop

Index	Lower bound	Upper bound
i	0	length - 1
j	0	2 * width - 1
k	0	breadth - 1

Connect

	Family Name	Index	Assigned Value
From Icon:	Qin	i	i
		j	j
		k	k
Output Port of the From Icon:	OUTPUT		
To Icon:	wildOne	k	k + 1
Input Port of the To Icon:	passIn	r	length - 1 - i
		s	j

When:

OK Cancel Apply Validate Form Print

Note that each instance in an Icon Family with height > 0 may have a family of Ports with height > 0. To specify connections between families of ports of two icons we use a 2-dimensional structure presented in the Connect table.

For each family of Ports in an Icon Family, the family indices of both the Icon Family and the Port Family are listed, and the user provides an integer expression which, when evaluated, assigns the value of the respective index. Each operand in the expressions is a literal number, a Formal GIP in the Prototype of the GUI Graph, or an index in an operator-defined nested loop. Each pass through the innermost loop establishes a single connection between two Ports that are identified by the respective values of the family indices. The user may specify a predicate expression that tells whether the connection should be made.

Nested loop

The Nested Loop is a [Family Tree table](#). Each expression for a Lower or Upper Bound is a literal number, a Formal GIP with height = 0 of the GUI Graph, or an index above the expression in the Nested Loop Table.

Connect table

For each Family Name, the Index and Assigned Value comprise a fixed table. The number of rows is equal to the respective family height. Specifically,

- For each icon, the family height is the height of the Icon Family Tree in the Icon's Call Form. In the Index column of the Connect Table, the GUI automatically enters the indices of the Icon Family Tree.
- For each Port, the family height is the height of the respective Port Family in the Icon's Prototype Form. In the Index column of the Connect Table, the GUI automatically enters the indices of the Port Family Tree.

In the expressions specifying Assigned Values, each operand is a literal number, a Formal GIP with height = 0 of the GUI Graph, or an index in the Nested Loop.

When expression

The user may define a boolean condition so called *when expression* that tells whether the connection should be made for the respective combination of assigned values of the indices in the Nested Loop. The default when expression is "true", i.e., the connection is made.

Each operand in the when expression is a literal number, a Formal GIP with height = 0 of the GUI Graph, or an index in the Nested Loop.

Additional constraints

No Port of any instance of an icon may be connected more than once.

No Port of any Icon may be connected if its family is associated with a Graph Port.

Exterior Forms

The Exterior Menu has forms related to the GUI Graph being edited. The user may select any of these forms to open and edit:

- [Prototype](#)
- [Port Association](#)
- [Banner](#)
- [Type List](#)
- [Included Graph List](#)

Prototype Form

The Prototype Form captures the interface information (i.e., the formal parameters) needed to create an instance of a Graph, Included Graph, Transition, or Place.

The Prototype Menu has the following items:

- New Ord Tran: Open a new Prototype Form for an Ordinary Transition.
- New Queue: Open a new Prototype Form for a non-standard queue.
- New GVar: Open a new Prototype Form for a non-standard graph variable.
- Operator-defined: Open a sub-menu of existing user-defined Transition and Place Prototype Forms. The user may select a Prototype Form from this menu to open and edit.
- System-defined: Open a sub-menu of system-defined Prototype Forms: Pack, Unpack, Queue, and GVar. The user may select one of these Prototype Forms to open read-only. The system-defined Queue and GVar Prototype Forms specify a single input port and a single output port (i.e., the port families have height 0).

To open the Prototype Form for the current GUI Graph, the user selects "Prototype" from the Exterior Menu.

To view the Prototype Form read-only for a given icon, the user clicks "Prototype form" in the icon's pop-up menu.

The Prototype Form depicted below includes all the information for all kinds of Prototype Form. In some cases, some of the information is not relevant and should be disabled. Where this is the case, we will so indicate.

PGMT

Prototype Form

Prototype Name: Entity Type:

Formal Type Arguments

Formal Name

Formal Mode Arguments

Token Height (Formal Name)	Base Type (Formal Name)

Formal Graph Instantiation Parameters (GIPs)

Name	Height (Number)	Base Type (Actual)

Input Ports

Family Name	Category	Token Ht	Base Type	Family Tree

Output Ports

Family Name	Category	Token Ht	Base Type	Family Tree

The Prototype Forms for special transitions, [Pack](#) and [Unpack](#), for standard queues, [Queue](#), and for standard graph variables, [GVar](#), are special cases of Prototype Forms with system-supplied entries in their fields. These four Prototype Forms are read-only. User-defined Place Prototype Forms are discussed in [Place Prototypes](#) section.

Prototype Name

The Prototype Name has to be unique among all Prototype Names in the GUI Graph. In particular, for a user-defined Prototype Form, the Prototype Name may not be Pack, Unpack, Queue, or GVar.

Entity Type

The Entity Type is "graph", "transition", "queue", "gvar", or "inclgraph", automatically set by the GUI according to the following rules:

- If the Prototype Form is the GUI Graph Prototype Form, opened via the Exterior Menu, the Entity Type is "graph". The Entity Type "graph" indicates that the Prototype Form is associated with the current GUI Graph. This not to be confused with the [Banner Form](#), where the user may choose between "Main Graph" and "Included Graph". The choice in the Banner Form indicates that the GUI Graph is to be instantiated either as a Main Graph (i.e., by the Command Program) or as an Included Graph (i.e., in another graph).
- If the Prototype Form is associated with one or more icons in the GUI Graph, the Entity Type reflects the kind of the associated icons:
 - If the Prototype Form is newly created for an Ord Tran, Queue, or GVar by selection from the Prototypes Menu, then the Entity Type is "transition", "queue", or "gvar", respectively.
 - If the Prototype Form is subsequently opened, the Entity Type is taken from the Parse Tree.
 - If the Prototype Form is opened by selection from an Included Graph Icon's pop-up Menu, then the Entity Type is not derived from any element of the parse tree. In this case, the Entity Type is "inclgraph". The Included Graph Icon's Prototype Form is read-only and, except for the Entity Type field, is identical to the Prototype Form of the underlying GUI Graph. This underlying GUI Graph is in the respective GSF whose full path is found in the [Included Graph List](#).

Formal Type Arguments

The Formal Type Arguments are unique among Variable Names in the Prototype Form and Type Names in the GUI Graph Prototype Form.

Formal Mode Arguments

In each Formal Mode Argument, the Token Height and Base Type are unique among Variable Names in the Prototype Form.

Formal Graph Instantiation Parameters (GIPs)

Each Formal GIP is a variable that represents a token of the specified Height and Base Type. In each Formal GIP

- the name is unique among Variable Names in the Prototype Form,
- the Base Type is either a [language-defined type](#) (int, float, ...) or a user-defined type that is defined in one of the files in the [Type List](#),
- the Height is an integer 0 or greater; if the specified Height is 0, then the GIP is a simple variable with the specified Base Type.

Ports

In each of the tables of Input Ports and Output Ports

- the Family Names are unique among Variable Names in the Prototype Form,
- the Category of each port is defined according to the following rules:
 - if the Prototype Entity Type is "transition", then the Category of each Port is "transition",
 - if the Prototype Entity Type is "queue" or "gvar", then the Category of each Port is "place",
 - if the Prototype Entity Type is "graph", then the Category may be either "transition" or "place", according to user choice.

Note: If in the [Banner Form](#) of the graph being edited, Main Graph is selected, then all port categories of the GUI Graph Prototype Form must be "place".

 - if the Prototype Entity is "inclgraph", then the Prototype Form is read-only.
- the Token Ht is an expression,
- each Base Type is either a language-defined type or a user-defined type that is defined in one of the files in the [Type List](#).
- In each Family Tree
 - each index is unique among the indices in its Family Tree and among all Variable Names in the Prototype Form. It is not necessary that indices be unique across different Family Trees in the Prototype Form.
 - in each expression specifying a Lower or Upper Bound, each operand is a literal number, a Formal GIP with height = 0 in the same Prototype, or an index in a row above the expression in the same Family Tree.
 - if the Prototype Entity Type is "transition", then the Height of the Family Tree (i.e., the number of rows in the Family Tree table) does not exceed the respective Token Ht.

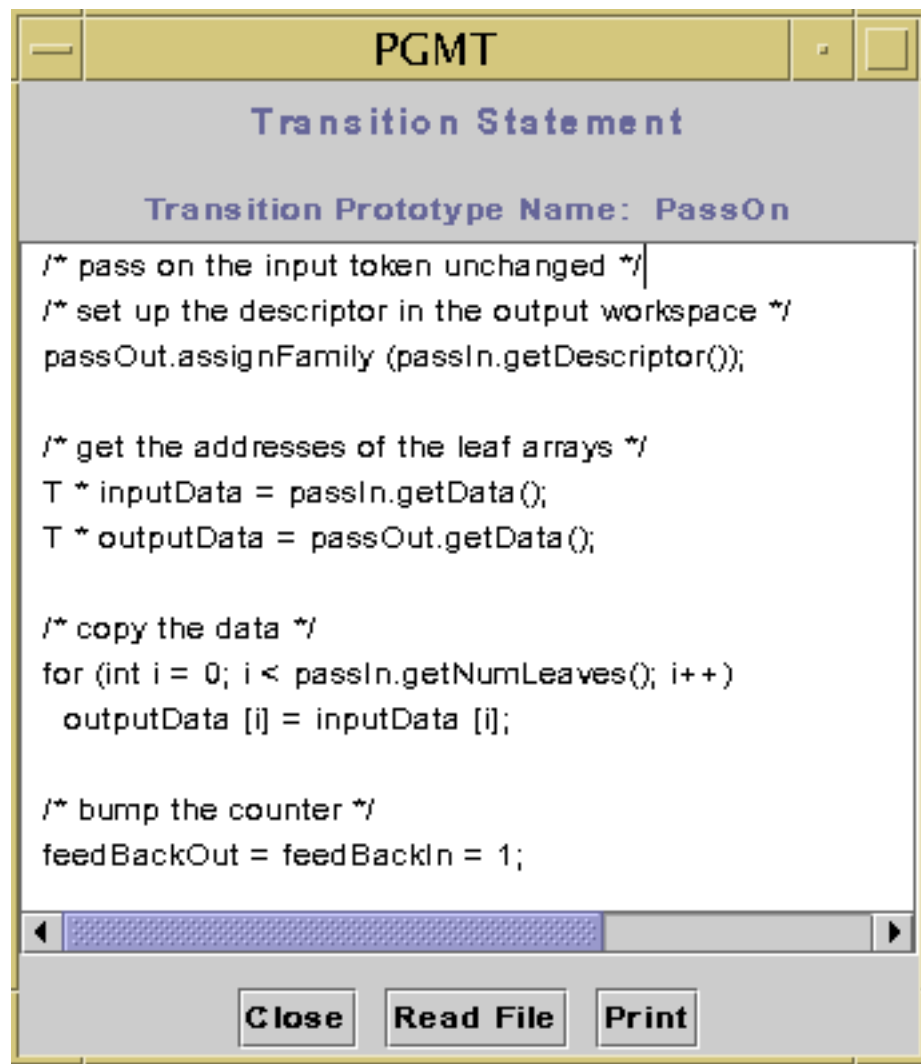
Body

If the Entity Type is either "transition" or "inclgraph", the Prototype Form contains the button *Open Body*. Clicking on this button will open a new window.

- If the Entity Type is "transition", the window displays the text of the Transition Statement, i.e., the body of a function which, when called, specifies the function of the transition. The user may assume that the Family Name of each Input Port and of each Output Port has been declared to be a variable that represents a token with the specified Height and Base Type. If the specified Height is 0, then the variable is a simple variable with the specified Base Type. Further, when the Transition Statement is called, the Name of each Input Port is initialized with the value of the token read from the respective Transition Input Port.

The window is read-only if the Prototype Form is read-only. If the window is editable, it contains the *Read File* button. Clicking on the button replaces the contents of the window with the contents of selected file.

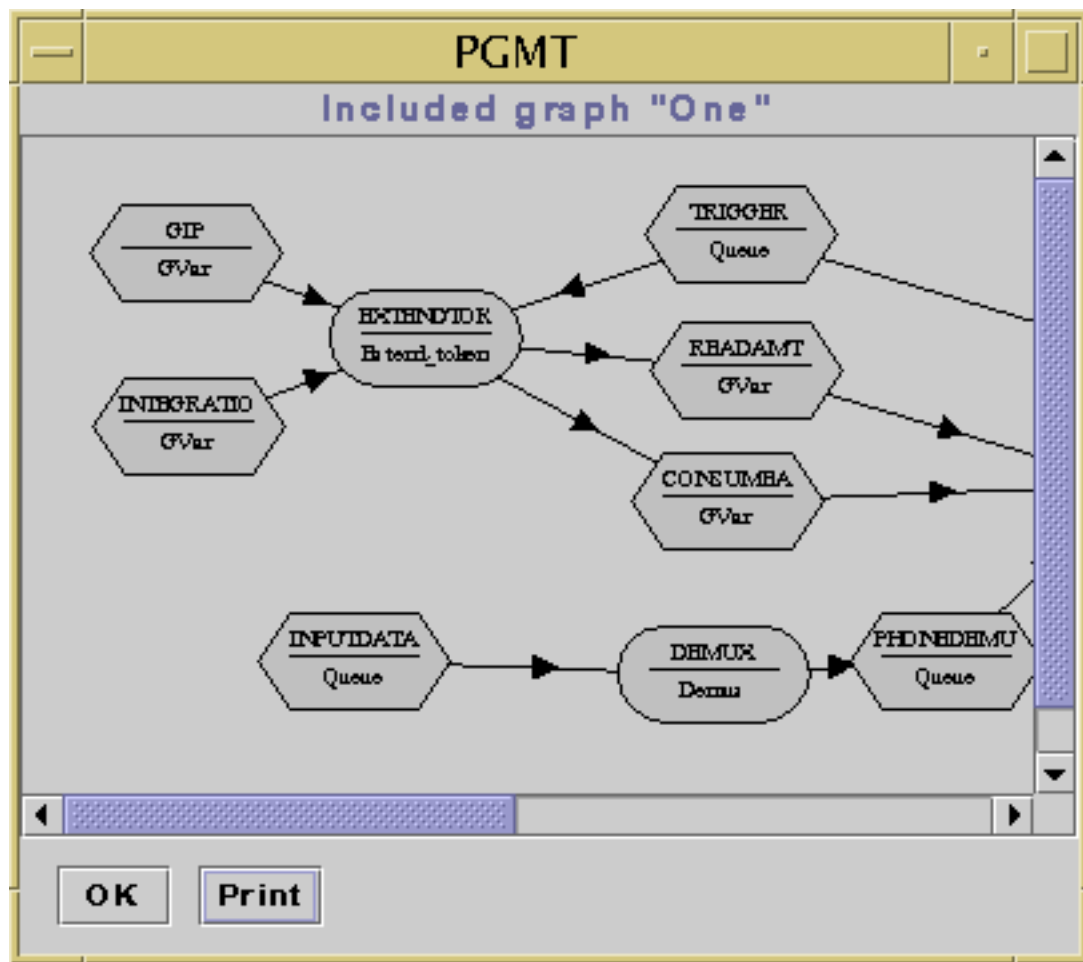
Example:



If the user modifies the contents of the Transition Statement window and wants to incorporate the changes made in the form (i.e. update the parse tree), he should close the form and click on either the *OK* or *Apply* button of the Prototype Form.

- If the Entity Type is "inclgraph", the window displays the graphic layout of the underlying GUI graph whose GSF is in the respective full path found in the [Included Graph List](#). This graphic layout window is read-only.

Example:



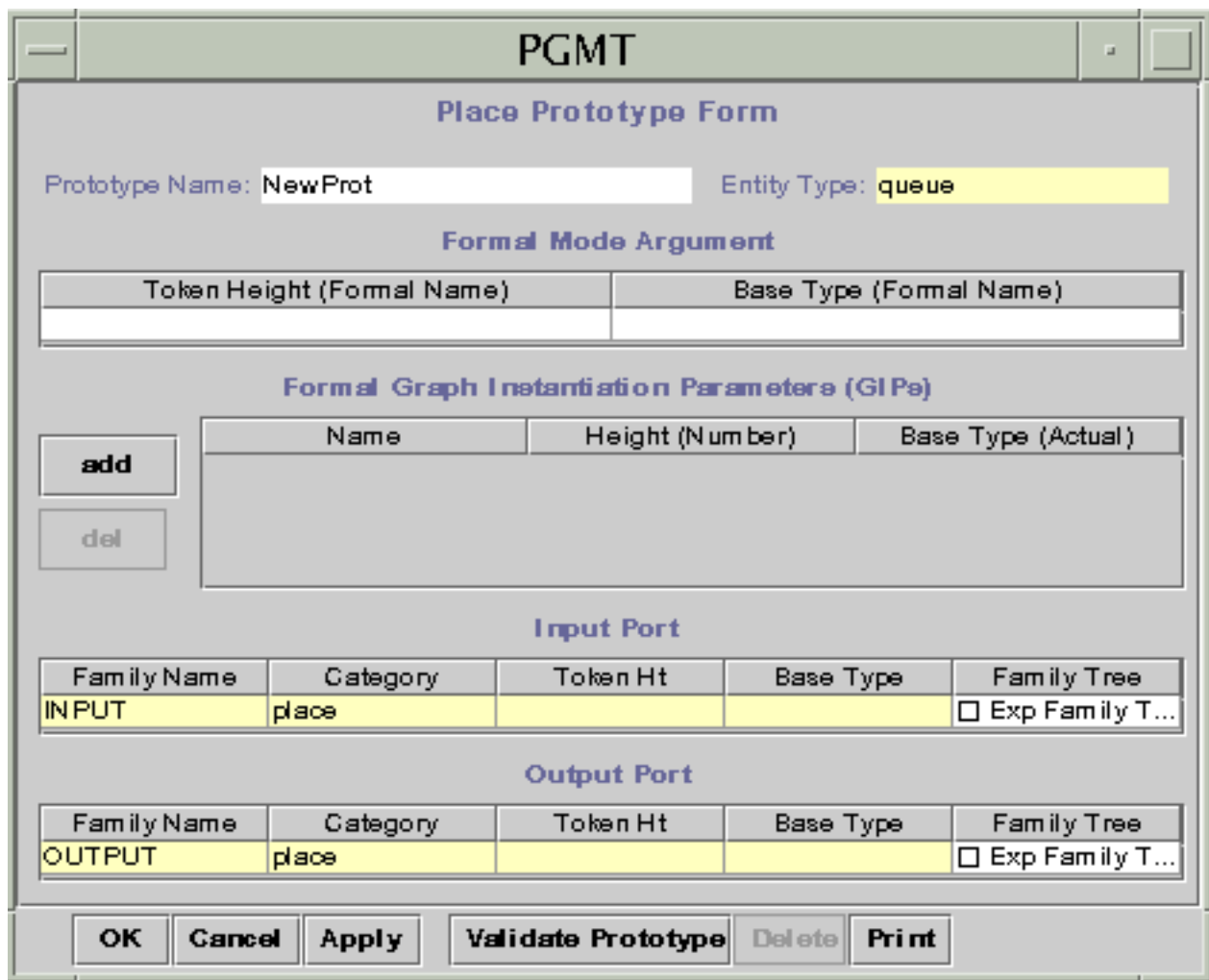
To edit the underlying GSF, the user opens the GSF in a separate GUI session.

Note: If the Entity Type is "queue", or "gvar", the *Open Body* button does not exist, because a place has no body.

Place Prototype Form

The Prototype Forms of non-standard queues and graph variables are user-defined. The user may open and define a Prototype Form for a non-standard queue or graph variable by selecting New Queue or New GVar from the Prototype Menu. The Place Prototype Form depicted here is a subset of the Prototype form depicted above. Except as noted, the Place Prototype Form conforms to the syntax and semantics rules for Prototype Forms.

Example of non-standard queue prototype form:



The screenshot shows the 'PGMT' window with the 'Place Prototype Form' tab selected. The 'Prototype Name' is 'NewProt' and the 'Entity Type' is 'queue'. Below this is the 'Formal Mode Argument' section with two empty fields for 'Token Height (Formal Name)' and 'Base Type (Formal Name)'. The 'Formal Graph Instantiation Parameters (GIPs)' section contains an 'add' button, a 'del' button, and a table with columns 'Name', 'Height (Number)', and 'Base Type (Actual)'. The 'Input Port' section has a table with columns 'Family Name', 'Category', 'Token Ht', 'Base Type', and 'Family Tree'. The 'Output Port' section has a similar table. At the bottom are buttons for 'OK', 'Cancel', 'Apply', 'Validate Prototype', 'Delete', and 'Print'.

PGMT

Place Prototype Form

Prototype Name: Entity Type:

Formal Mode Argument

Token Height (Formal Name)	Base Type (Formal Name)
<input type="text"/>	<input type="text"/>

Formal Graph Instantiation Parameters (GIPs)

Name	Height (Number)	Base Type (Actual)
<input type="text"/>	<input type="text"/>	<input type="text"/>

Input Port

Family Name	Category	Token Ht	Base Type	Family Tree
INPUT	place	<input type="text"/>	<input type="text"/>	<input type="checkbox"/> Exp Family T...

Output Port

Family Name	Category	Token Ht	Base Type	Family Tree
OUTPUT	place	<input type="text"/>	<input type="text"/>	<input type="checkbox"/> Exp Family T...

Entity Type

The Entity Type is either "queue" or "gvar".

Input and Output Ports

In each of the tables of Input Ports and Output Ports, there is exactly one Port, and the tables are fixed with one row each.

- The Family Names are "INPUT" and "OUTPUT".
- The Category is "place".
- For the Token Ht and Base Type, the GUI repeats the Token Height and Base Type that appear in the Formal Mode Argument. These specify the mode of the tokens stored in the place.
- The user may edit either or both of the Family Trees in the Input and Output Port tables.

Graph Port Associations Form

The Graph Port Associations Form identifies each graph port family with a family of ports of an icon in the graph.

Example:

PGMT

Graph Port Associations

Graph Input Ports

Graph Input Port Family Name	Icon Family Name	Icon Input Port Family Name
INTEGRATION_GP	INTEGRATION	INPUT
ANGLE_GP	ANGLE	INPUT
LOFREQ_GP	LOFREQ	INPUT
WINDOWSIZE_GP	WINDOWSIZE	INPUT
INPUTDATA_GP	INPUTDATA	INPUT

Graph Output Ports

Graph Output Port Family Name	Icon Family Name	Icon Output Port Family Name
OUTPUTDATA_GP	OUTPUTDATA	OUTPUT

OK Cancel Apply Validate Form Print

The GUI automatically fills in the left-most column with the Graph Port Family Names of the graph, as listed in the respective expandable table of the Graph Prototype Form.

For each Graph Port Family the user enters the Icon Family Name and Port Family Name to be associated with the Graph Port Family. The following conditions must be respected:

- The Icon Family Name is the Family Name of some icon in the graph.
- The Port Family Name is the Family Name of some port in the Icon's Prototype Form.
- No Icon Port Family is associated with more than one Graph Port Family.
- The category of each Graph Port and the associated Icon Port are the same category (as specified in the respective prototype forms).
- The direction of the graph port and the icon port are the same (i.e., graph input ports are associated only with icon input ports, and graph output ports are associated only with icon output ports).

All of the above conditions are validated by the GUI. Additionally, the form has to satisfy the following two conditions that cannot be, in general, validated by the GUI.

- The mode (i.e., token height and base type) of the Graph Port and the mode of the associated Icon Port are the same.
- The family tree of each graph port family (specified in the Graph Prototype Form) matches the family tree of the associated icon port family (specified in the icon's Prototype Form).

Banner Form

The Graph Banner Form is selected from the Exterior Menu. The user provides some general information about the current graph state file:

- file name,
- author,
- version, and
- purpose

The user also specifies whether the current graph is a main graph that is called by a command program, or an included graph that will be incorporated into another graph.

Example:

The first line is read only and is automatically filled in by the gui.

Type List Form

Type List Form is used to identify user-defined types. Each entry is composed of a type name and a full path of the header file defining this type name, separated by "@".

Each header file can contain definitions of several user-defined types. However, the only known to the GUI type names are the names explicitly mentioned in one of the entries of the Type List Form.

If an entry in the form is selected, then the user can use the browse button to select a header file for this entry.

Example:



The entries in the table are automatically sorted by the GUI when either the *OK* or *Apply* button is clicked.

A type name cannot be a direct instantiation of a class template. To use a type instance of a class template, the user has to define an explicit type (i.e., without template arguments) via typedef in the source defining the user-defined classes. More information on this subject can be found in [Mtool/PGMT Integration](#) document.

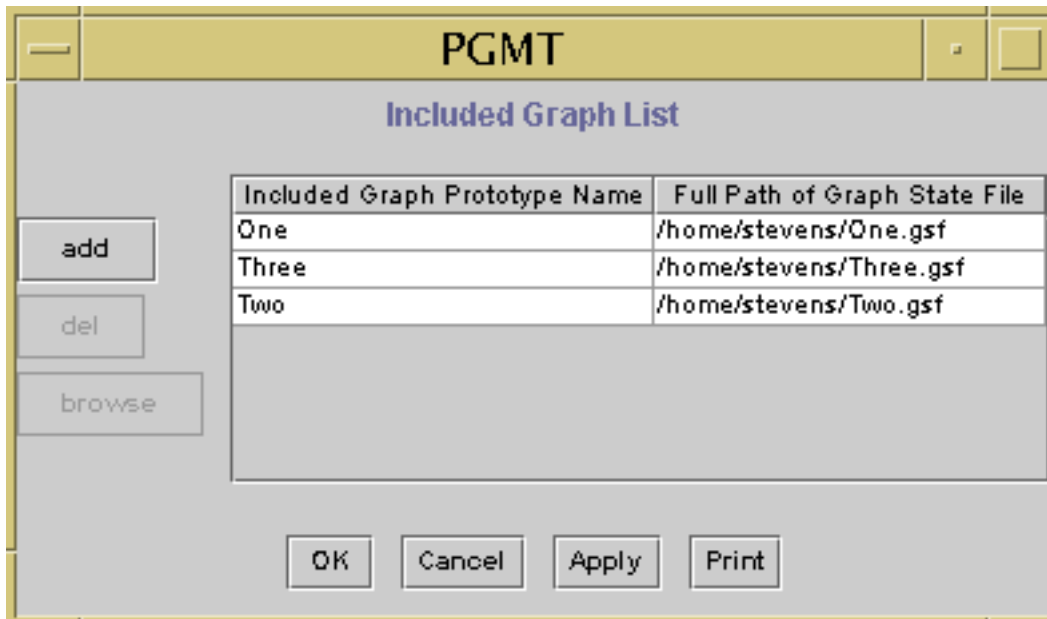
Included Graph List Form

The form lists all graph prototype names used in the current graph. Each row of the table contains the name of an included graph prototype and the full path of the graph state file containing its definition.

The full path of graph state file can be selected via a file browser.

The entries are sorted by the GUI.

Example:



The entries in the table are automatically sorted by the GUI when either of *OK* or *Apply* buttons is clicked.

No GUI Graph may contain itself as an Included Graph, either directly or indirectly.

Editing

Editing forms

To maintain coherence of the parse tree, while the user is editing a form, the GUI blocks the user from doing anything that either results in a change of the parse tree or has the potential to do so. In particular, the user may edit no more than one form at a time. While editing a given form, the user may view any other form, but may not edit it. Further, while editing a given form, the user may not add icons or arcs and may not move any icons or in any other way cause the graph display to change. To complete editing a form in order to perform another action, the user finishes editing the current form by clicking on one of the three buttons, OK, Apply, or Cancel in the current form.

Clicking on the Apply Button performs the syntax check of every field in the form and reports all errors to the user. A blank field is acceptable. If there are no errors, the GUI updates the parse tree to incorporate changes made in the form.

Clicking on the OK Button performs the actions described for Apply and then closes the form. Clicking on the Cancel Button closes the form without checking against the syntax rules and without making any changes to the parse tree.

Some of the forms contain the Validate Form Button. Clicking on this button performs local semantic checks. Blank fields are not acceptable. To make a full [validation](#), the user should select Validate Graph from the Translation menu.

If the user is editing a form and opens, closes, or saves a file, or creates a new file, the GUI first displays a message asking the user to close the form.

Some information that the user enters in one form can also appear in other forms. When the user edits one form and clicks either Apply or OK, the GUI closes all opened forms containing the changed information.

Editing graphic elements

Editing graphic elements can be done in four different modes.

Selecting, deselecting, moving, copying, cutting, and pasting graphic elements can be done in the [select mode](#). To enter the select mode, choose Select from the Nodes menu.

Inserting new icons is done in the *icon insertion mode*. To enter the icon insertion mode, choose Transition, Place, or Included Graph from the Nodes menu to insert a transition icon, a place icon, or an included graph icon respectively.

Inserting a new arc is done in the [arc insertion mode](#). To enter the arc insertion mode choose Arc from the Nodes menu.

Changing the shape of an arc is done in the [arc bends mode](#). To enter the arc bends mode choose Arc Bends from the Nodes menu.

Select Mode

You can select one or several objects at once. When one or more objects are selected, the display color changes.

To select an object

To select an object, click it.

Selecting an object deselects any currently selected objects.

To select several objects

You can select several objects at once, or you can add an object to an existing selection.

To select several objects, point outside the objects and drag diagonally to draw a selection border around them.

To add an object to a selection, hold down Shift while you click the object.

To add several objects to a selection, hold down Shift and drag diagonally to draw a selection border.

To deselect objects

You can deselect one object, several objects, or all selected objects at once.

To deselect the only object selected, click outside the object.

To deselect one of several selected objects, Shift-click the object.

To deselect all selected objects, click on the frame away from any objects.

To move objects

You can move objects in any direction, or you can constrain the direction to horizontal or vertical.

To move an object, select the object and drag it.

To move the object horizontally or vertically, hold down CTRL and Shift while you drag the object.

To move several objects, select the objects and hold down Shift while you drag them.

To move several objects horizontally or vertically, select the objects and hold down Ctrl and Shift while you drag them.

Arc Insertion Mode

You can draw an arc in either direction between any two icons if at least one of them is an included graph icon, or between

a place icon and a transition icon.

Arc Bends Mode

To add a new bend, click on an arc segment and pull it out.

To remove an existing bend, click with the right button on the bend.

Validation

Validating Forms

Any form that is being edited, may be validated by clicking on the Validate Form Button. All syntactic and semantic errors will be displayed. The validating process can be stopped at any time by clicking on the Stop Validation Button.

In many situations when a syntactic error is detected, a meaningful semantic analysis is impossible and it will be disabled.

Observe that clicking on the Apply Button or OK Button will only perform syntactic checks in a given form. This allows to save a graph with partially filled forms.

Validating the Graph

At any time that no form is being edited, the user may select Validate Graph in the Translation Menu. In response, the GUI will check all forms of the current GUI graph against the semantics rules and report errors to the user.

In some situations, reporting all errors is impossible. For example, if an invalid prototype is found, validating Call Forms does not make sense and further validation will be stopped.

There are also situations when the GUI does not dispose sufficient information to validate a form, and informs the user by displaying error warnings. For example, if the user uses a data type different from predefined types in C++, the GUI displays the message: "GUI cannot validate the type "t"". During the [batch execution](#), the user can inhibit displaying warnings with the option "-w".

Translation

At any time that no form is being edited, the user may select Output C++ in the Translation Menu. In response, the GUI will first validate the current GUI graph. If there are no errors, the GUI will generate the C++ code for the current GUI graph according to the [Translator Specification](#).

Before saving the C++ code to a file, the GUI asks the user for a name of this file. The default name is n.h where n is the Graph Class Name (specified by the user in the Prototype Name in the Graph Prototype Form).

Miscellaneous

Batch execution

The following parameters are allowed:

-c file.gsf	Translate the gsf file to C++ code.
-d directory	Specify the C++ destination directory.
-h	Display the help information.
-v file.gsf	Validate the gsf file.
-V	Print the version information.
-w	Inhibit all warning messages.

Support for C++

Some of the fields used in Prototype and Call Forms require usage of types and expressions written in a programming language used by the PGM software. Currently, PGM supports the programming language C++. GUI accepts a subset of C++ data types and expressions. Below, we define the syntax of different C++ constructions used by the GUI.

types

The following C++ types are supported:

char	short	int	long
unsigned char	unsigned short	unsigned int	unsigned long
float	double	long double	

literal numbers

GUI accepts integer and double literals. An integer literal is a sequence of decimal digits. A double literal is a sequence of decimal digits, possibly empty, followed by a dot and a sequence of decimal digits, possibly empty.

variable names

A variable name is a sequence of letters, digits or underscores starting with a letter or an underscore. There are a number of identifiers that have a special meaning in gsf files and cannot be used as variable names. They are listed in Appendix A.

expressions

An expression is one of the following:

- variable name
- literal number

- unary expression, <operator> <expression>, where <operator> is one of the following:

+ - & ++ -- ~ !

- unary expression, <expression> <operator>, where <operator> is one of the following:

* ++ --

- binary expression, <expression> <operator> <expression>, where <operator> is one of the following:

+ - * / % ^ > < >= <= == !=
&& || << >> & | ->

- indexed array
- function call

Format of GSF files

The following identifiers are used as keywords and may not be used as variable names:

arc author banner bends capsule
category exterior filename fmly gips
graphtype gvar inclgraph initval inport
input leaf location outport output
place prototype purpose queue revision
transition trstmt type unsigned when

The GSF grammar is defined below using the BNF notation:

Non-terminal symbols:

Pgsf ::= capsule { Banner Exterior Specs }
Arc ::= arc RangeSeq BendSeq { Link }
Args ::= null | Expr | Args COMMA Expr
Assoc ::= Name <=> label . Name ; |
 Assoc Name <=> label . Name ;
Banner ::= banner { filename CodeString author CodeString
 revision CodeString purpose CodeString
 graphtype number ; }
BaseAux ::= null | BaseAux , BaseType
BasePrim ::= null | BaseType BaseAux
BaseType ::= Name TypeSeq |
 unsigned Name TypeSeq | // C++ dependent
 unsigned TypeSeq // C++ dependent
BendSeq ::= null | bends PtSeq
Bindings ::= Name = Value ; | Bindings Name = Value ;
Cat ::= null | category transition | category place

```
ClassCall ::= Name TypeSeq
ClassName ::= Name TempPrim
Connect  ::= Name ExSeq . Name ExSeq
Expr     ::=                               // C++ dependent
    Number | FloatNumber | Name | ( Expr ) |
    Expr + Expr | Expr - Expr | Expr * Expr |
    Expr / Expr | Expr % Expr | Expr ^ Expr |
    Expr > Expr | Expr < Expr | Expr >= Expr |
    Expr <= Expr | Expr == Expr | Expr != Expr |
    Expr && Expr | Expr || Expr | Expr << Expr |
    Expr >> Expr | + Expr | - Expr | * Expr |
    & Expr | ++ Expr | Expr ++ | -- Expr |
    Expr -- | ~ Expr | Expr ? Expr : Expr |
    Expr & Expr | Expr | Expr | ! Expr |
    Expr [ Expr ] | Expr ( Args ) | Expr -> Expr
ExSeq    ::= null | ExSeq [ Expr ]
Exterior ::= exterior { Prototype Pass Ptype }
FamInit  ::= null | fmly ( Mode ModeAux )
FamName  ::= null | fmly ( ForArg ForArgAux )
ForArg   ::= < Name , Name >
ForArgAux ::= null | ForArgAux , ForArg
Gips     ::= Name : Mode ; | Gips Name : Mode ;
Gos      ::= null | gips { Gips }
Gparam   ::= null | gips { Bindings }
InclForm ::= inclgraph { InclIden }
InclIden ::= Name @ PathName ; | InclIden Name @ PathName ;
INP      ::= null | inport { Port }
Instance ::= label RangeSeq location < Location > =
    Kind ClassCall FamInit Params ;
Iparam   ::= null | initval { Value }
Kind     ::= transition | place | incgraph
Link     ::= Connect -> Connect | Connect -> Connect when Expr
Location ::= number , number
Mode     ::= < Expr , BaseType >
ModeAux  ::= null | ModeAux , Mode
NestAux  ::= Expr | { NestPrim }
NestedString ::= { NestPrim }
NestPrim ::= null | NestPrim NestAux
NodeProto ::= transition { Prototype } | gvar { Prototype } |
    queue { Prototype }
OUTP     ::= null | outport { Port }
Params   ::= null | { Gparam Iparam }
Pass     ::= null | Pass PortAssoc
Ptype    ::= null | InclForm
Points   ::= number , number
Port     ::= Name Mode RangeSeq Cat ; |
    Port Name Mode RangeSeq Cat ;
PortAssoc ::= input { Assoc } | output { Assoc }
Prototype ::= prototype ClassName FamName ;
    Gos INP OUTP TrStmt
```

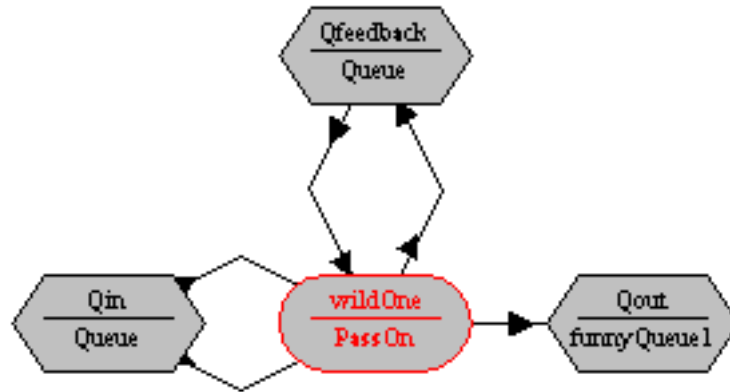
PtSeq ::= < Points > | PtSeq < Points >
Ptype ::= null | TypeForm
Range ::= Expr : Name : Expr
RangeSeq ::= null | RangeSeq [Range]
Specs ::= null | Specs NodeProto | Specs Instance |
Specs Arc
TempAux ::= null | TempAux , Name
TempPrim ::= null | < Name TempAux >
TrStmt ::= trstmt CodeString
TypeForm ::= type { TypeIden ; }
TypeIden ::= Name @ PathName ; | TypeIden Name @ PathName ;
TypeSeq ::= null | < BaseType BaseAux >
Value ::= RangeSeq leaf [Expr] | NestedString

Terminal symbols:

CodeString ::= a string that starts with '{:', ends with ':}',
and does not contain ':'
FloatNumber ::= a string containing one or more digits with
one dot placed anywhere
Name ::= a string that starts with a letter or underscore,
followed by zero or more letters, numbers, or
underscores
null ::= empty string
Number ::= a string containing one or more digits
PathName ::= a string that starts with '"', ends with '"',
and does not contain '"'

Example

This example was designed more to illustrate different forms in the GUI than to do model a real application.



Every icon represents a family with height ≥ 0 .

- Qin is a family of queues with height = 3.
- QfeedBack and wildOne each have family height = 1.
- Qout has family height 0 and is thus a single queue. Qout is a queue with a family of input ports with height = 2.

PassOn defines the prototype for the transition wildOne. For each transition in the family called wildOne, there is a family of input ports called passIn with height = 0. These ports are connected in a strange way to the output ports of the queues in the family Qin. When the second index j is even (???), the ports are connected in the order specified by the other indices i and k . When the second index j is odd (???), the ports are connected so that the order is reversed for the first index i .

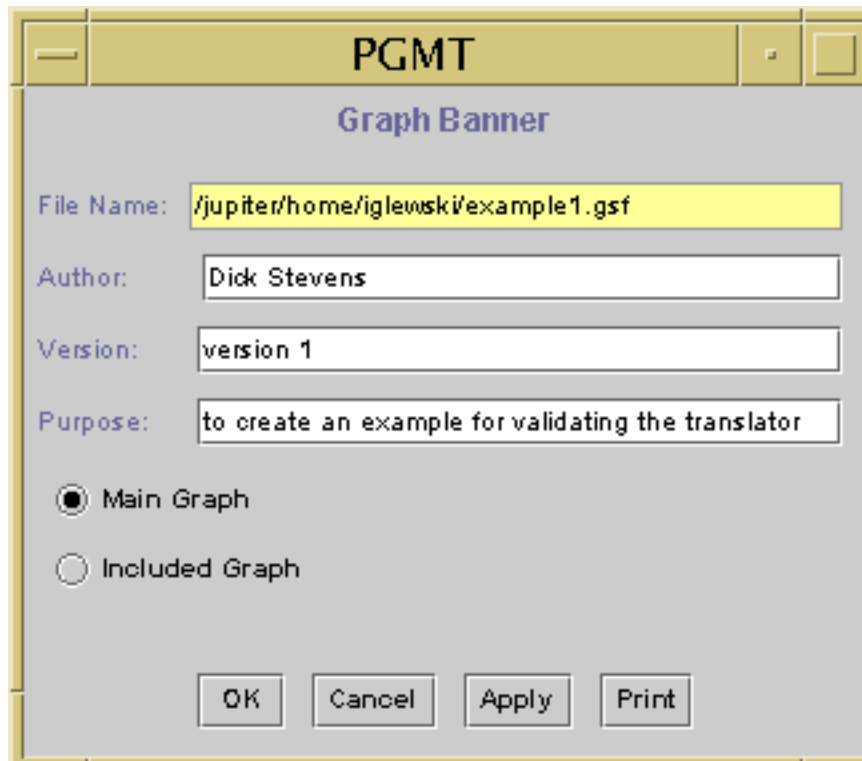
The tokens in the queues Qin and Qout have base type char. The tokens in the queues QfeedBack have base type unsigned int. Each transition in the family wildOne is connected to a queue in QfeedBack, which serves to count the executions of the transition. Each queue in QfeedBack is initialized with a token whose value is the index of the queue in the family.

In each execution of wildOne, one token is read from each input port connected to a queue in the family Qin. Those tokens are then assembled into a single token that has family height 2 and produced to the output port passOut, which is connected to one of the input ports of Qout. When a transition fires, it reads an unsigned integer from its member of the family of queues in QfeedBack, adds one, and produces the new value to the same queue.

Qout is a single queue with a family of input ports. Thus, Qout stores the tokens in the order that the wildOne transitions execute.

Exterior Forms

Banner Form



The image shows a dialog box titled "PGMT" with a subtitle "Graph Banner". It contains four text input fields: "File Name:" with the value "/jupiter/home/iglewski/example1.gsf", "Author:" with "Dick Stevens", "Version:" with "version 1", and "Purpose:" with "to create an example for validating the translator". Below these fields are two radio buttons: "Main Graph" (selected) and "Included Graph". At the bottom are four buttons: "OK", "Cancel", "Apply", and "Print".

PGMT

Graph Banner

File Name: /jupiter/home/iglewski/example1.gsf

Author: Dick Stevens

Version: version 1

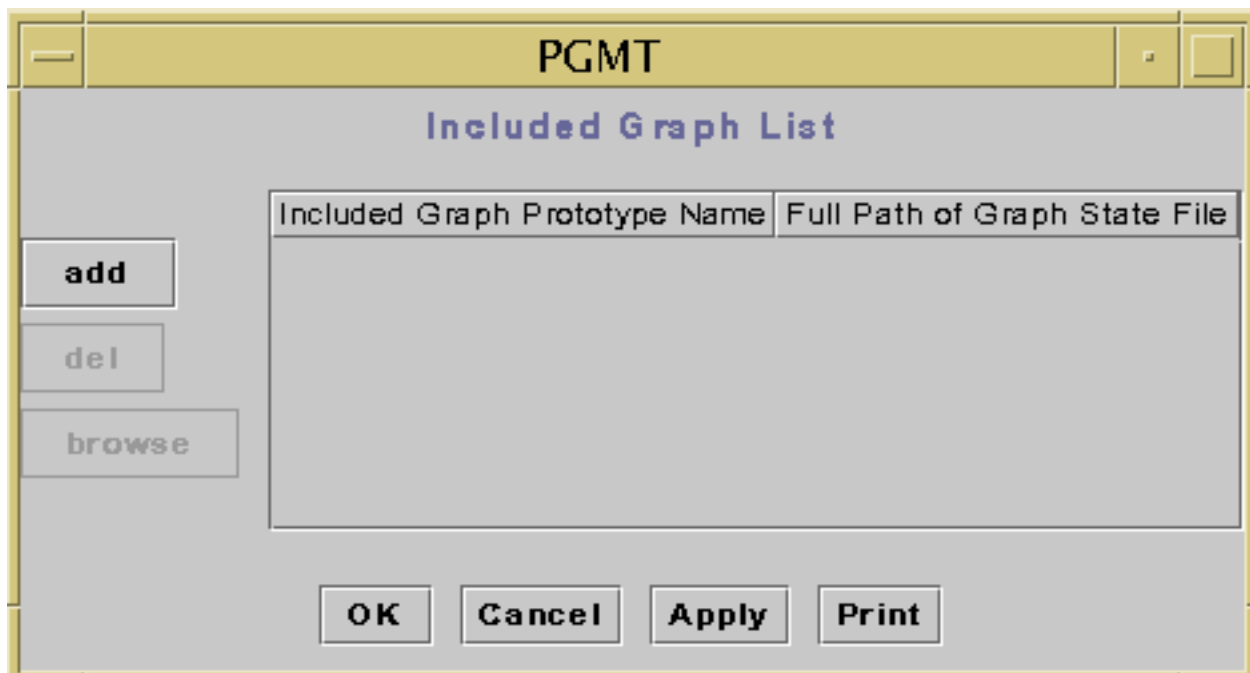
Purpose: to create an example for validating the translator

☒ Main Graph

☐ Included Graph

OK Cancel Apply Print

Included Graph List Form



The image shows a dialog box titled "PGMT" with a subtitle "Included Graph List". On the left side, there are three buttons: "add", "del", and "browse". To the right of these buttons is a large table with two columns: "Included Graph Prototype Name" and "Full Path of Graph State File". The table is currently empty. At the bottom are four buttons: "OK", "Cancel", "Apply", and "Print".

PGMT

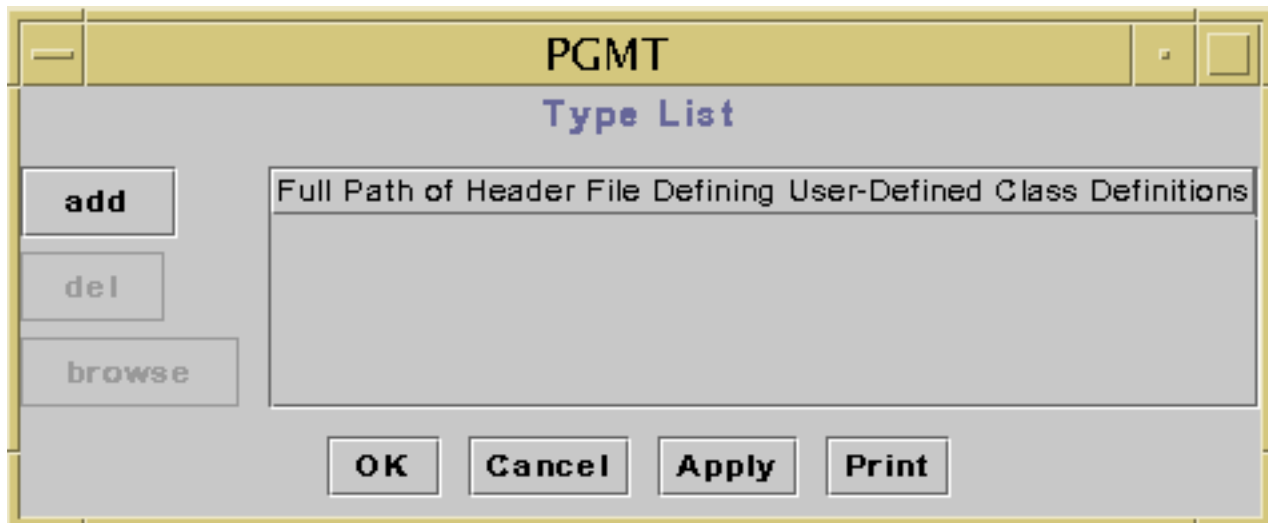
Included Graph List

add del browse

Included Graph Prototype Name	Full Path of Graph State File
-------------------------------	-------------------------------

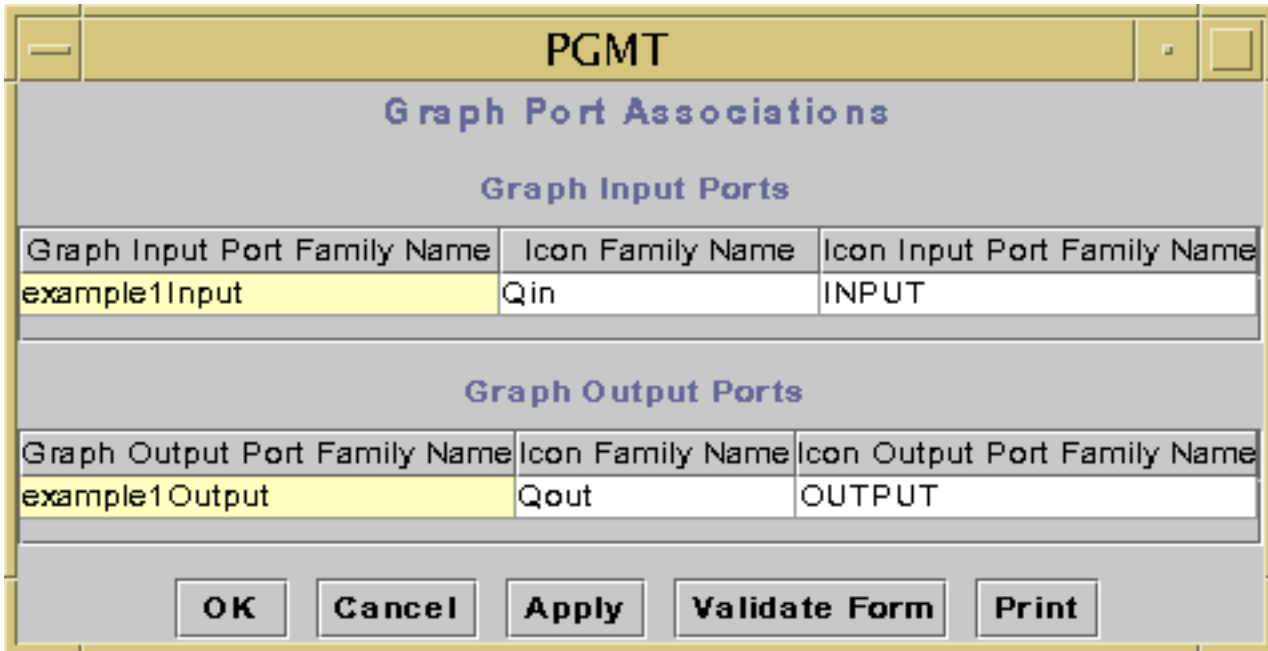
OK Cancel Apply Print

Type List Form



The image shows a dialog box titled "PGMT" with a subtitle "Type List". On the left side, there are three buttons: "add", "del", and "browse". To the right of these buttons is a large text area with the placeholder text "Full Path of Header File Defining User-Defined Class Definitions". At the bottom of the dialog, there are four buttons: "OK", "Cancel", "Apply", and "Print".

Graph Port Associations Form



The image shows a dialog box titled "PGMT" with a subtitle "Graph Port Associations". It contains two tables. The first table is titled "Graph Input Ports" and has three columns: "Graph Input Port Family Name", "Icon Family Name", and "Icon Input Port Family Name". The second table is titled "Graph Output Ports" and has the same three columns. Both tables have one row highlighted in yellow. At the bottom of the dialog, there are five buttons: "OK", "Cancel", "Apply", "Validate Form", and "Print".

Graph Input Port Family Name	Icon Family Name	Icon Input Port Family Name
example1Input	Qin	INPUT

Graph Output Port Family Name	Icon Family Name	Icon Output Port Family Name
example1Output	Qout	OUTPUT

Graph Prototype Form

PGMT

Prototype Form

Prototype Name: Entity Type:

Formal Type Arguments

add

del

Formal Name

Formal Mode Arguments

add

del

Token Height (Formal Name)	Base Type (Formal Name)
----------------------------	-------------------------

Formal Graph Instantiation Parameters (GIPs)

add

del

Name	Height (Number)	Base Type (Actual)
length	0	unsigned int
width	0	unsigned int
breadth	0	unsigned int

Input Ports

add

del

Family Name	Category	Token Ht	Base Type	Family Tree
example1Input	place	0	char	<input type="checkbox"/> Exp Famil...

Output Ports

add

del

Family Name	Category	Token Ht	Base Type	Family Tree
example1Out...	place	2	char	<input type="checkbox"/> Exp Famil...

Open Body

OK

Cancel

Apply

Validate Prototype

Print

Family Tree (example1Input)

Family Tree

add

del

Index	Lower Bound	Upper Bound
i	0	length - 1
j	0	2 * width - 1
k	0	breadth - 1

OK **Cancel** **Print**

Family Tree (example1Output)

Family Tree

add

del

Index	Lower Bound	Upper Bound
-------	-------------	-------------

OK **Cancel** **Print**

Icon Prototype Forms

Transition Prototype Form for PassOn

PGMT

Prototype Form

Prototype Name: PassOn

Entity Type: transition

Formal Type Arguments

add

del

Formal Name
T

Formal Mode Arguments

add

del

Token Height (Formal Name)	Base Type (Formal Name)
----------------------------	-------------------------

Formal Graph Instantiation Parameters (GIPs)

add

del

Name	Height (Number)	Base Type (Actual)
how_long	0	unsigned int
how_wide	0	unsigned int

Input Ports

add

del

Family Name	Category	Token Ht	Base Type	Family Tree
passIn	transition	0	T	<input type="checkbox"/> Exp Famil...
feedBackIn	transition	0	unsigned int	<input type="checkbox"/> Exp Famil...

Output Ports

add

del

Family Name	Category	Token Ht	Base Type	Family Tree
passOut	transition	2	T	<input type="checkbox"/> Exp Famil...
feedBackOut	transition	0	unsigned int	<input type="checkbox"/> Exp Famil...

Open Body

OK

Cancel

Apply

Validate Prototype

Delete

Print

Family Tree (passIn)

The dialog box titled "Family Tree" contains a table with three columns: "Index", "Lower Bound", and "Upper Bound". The table has two rows of data. To the left of the table are two buttons: "add" and "del". At the bottom are three buttons: "OK", "Cancel", and "Print".

Index	Lower Bound	Upper Bound
r	0	how_long - 1
s	0	how_wide - 1

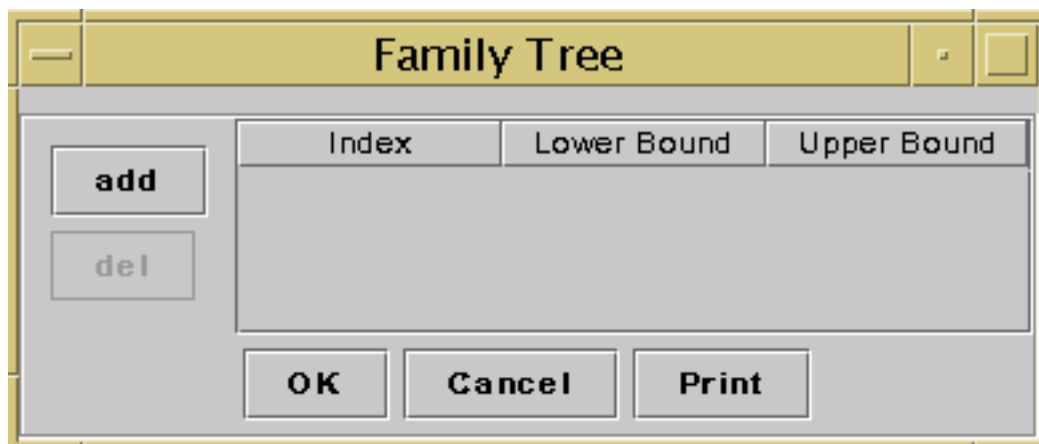
Family Tree (feedBackIn)

The dialog box titled "Family Tree" is identical in layout to the first one, but the table is empty. It has columns "Index", "Lower Bound", and "Upper Bound". The "add", "del", "OK", "Cancel", and "Print" buttons are present.

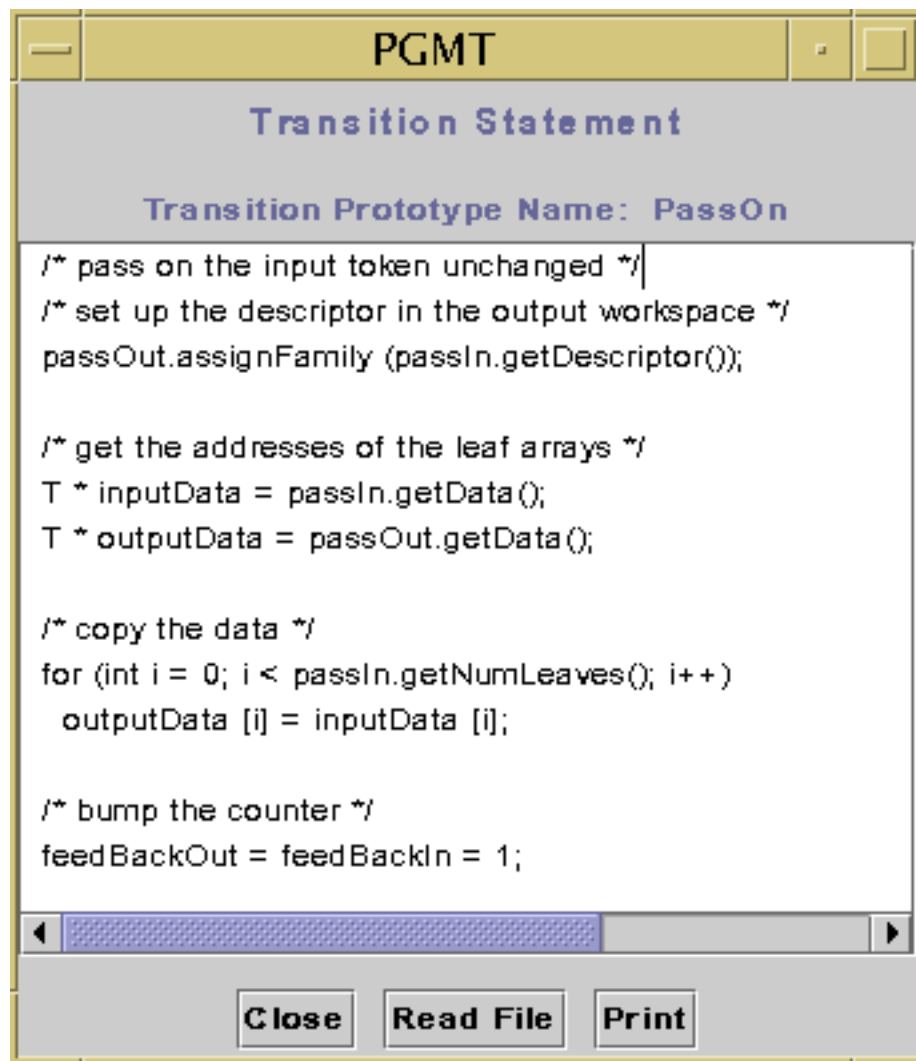
Family Tree (passOut)

The dialog box titled "Family Tree" is identical in layout to the previous ones, but the table is empty. It has columns "Index", "Lower Bound", and "Upper Bound". The "add", "del", "OK", "Cancel", and "Print" buttons are present.

Family Tree (feedBackOut)



Transition Statement



Queue Prototype Form for funnyQueue1

PGMT

Place Prototype Form

Prototype Name: Entity Type:

Formal Mode Argument

Token Height (Formal Name)	Base Type (Formal Name)
height	base_type

Formal Graph Instantiation Parameters (GIPs)

Name	Height (Number)	Base Type (Actual)
how_broad	0	unsigned int

Input Port

Family Name	Category	Token Ht	Base Type	Family Tree
INPUT	place	height	base_type	<input type="checkbox"/> Exp Family Tree

Output Port

Family Name	Category	Token Ht	Base Type	Family Tree
OUTPUT	place	height	base_type	<input type="checkbox"/> Exp Family Tree

Family Tree (INPUT)

Family Tree

Index	Lower Bound	Upper Bound
i	0	how_broad

Family Tree (OUTPUT)

Family Tree

add

del

Index	Lower Bound	Upper Bound

OK

Cancel

Print

Call Forms

Call Form for transition icon wildOne

PGMT

Call Form

Icon Family Name: wildOne

Prototype Name: PassOn

Icon Family Tree

add

del

Index	Lower bound	Upper bound
k	1	breadth

Actual Type Arguments

Formal Type	Base Type
T	char

Actual Mode Arguments

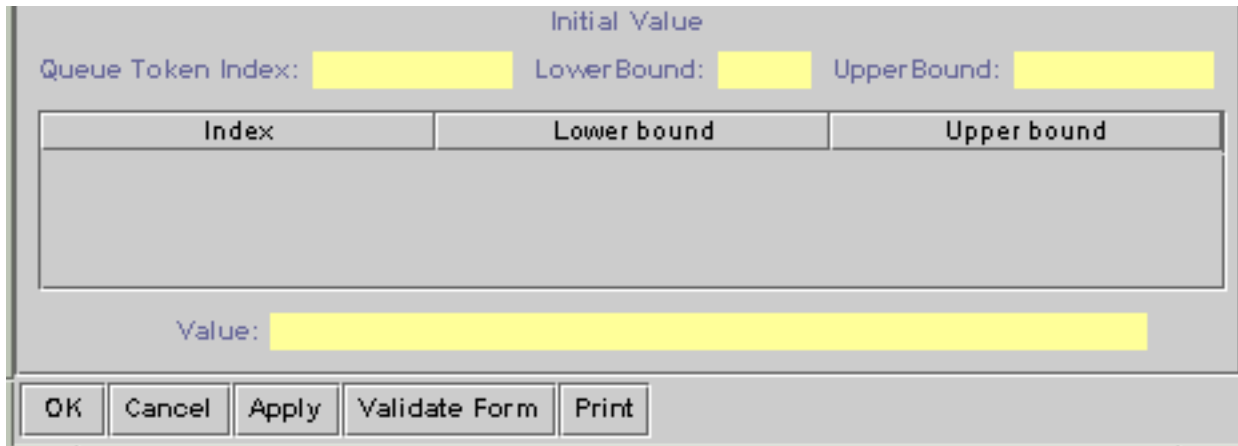
Formal Height	Actual Height	Formal Base Type	Actual Base Type
---------------	---------------	------------------	------------------

Actual GIP Bindings

Formal GIP	Family Tree	Value
how_wide	<input type="checkbox"/> Fix Family Tree	2 * width
how_long	<input type="checkbox"/> Fix Family Tree	length

Initial Value

Queue Token Index: LowerBound: UpperBound:

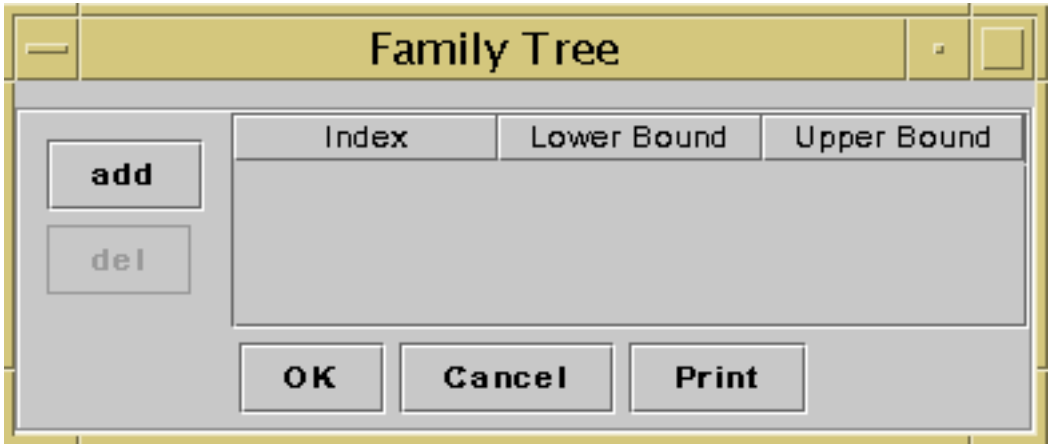


A dialog box titled "Initial Value". It contains three input fields: "Queue Token Index:", "LowerBound:", and "UpperBound:". Below these is a table with three columns: "Index", "Lower bound", and "Upper bound". The table is currently empty. Below the table is a "Value:" label followed by a long input field. At the bottom are five buttons: "OK", "Cancel", "Apply", "Validate Form", and "Print".

Index	Lower bound	Upper bound
-------	-------------	-------------

Note: The Initial Value section is disabled, because there is no initial value for a Transition Call Form.

Family Tree (how_wide)



A dialog box titled "Family Tree". It has a table with three columns: "Index", "Lower Bound", and "Upper Bound". To the left of the table are two buttons: "add" and "del". Below the table are three buttons: "OK", "Cancel", and "Print".

Index	Lower Bound	Upper Bound
-------	-------------	-------------

Note: *length* is a graph formal GIP so the Family Tree table for *how_long* is ignored and cannot be edited.

Call Form for queue icon Qin

<http://w3.uqah.quebec.ca/iglewski/AIT/GUI/last/pgmtHelp/guihelp/example.html> (12 of 26) [7/19/2002 4:50:47 PM]

PGMT

Call Form

Icon Family Name: Qfeedback

Prototype Name: Queue

Icon Family Tree

add

Index	Lower bound	Upper bound
k	1	breadth

del

Actual Type Arguments

Formal Type	Base Type
-------------	-----------

Actual Mode Arguments

Formal Height	Actual Height	Formal Base Type	Actual Base Type
height	0	base_type	unsigned int

Actual GIP Bindings

Formal GIP	Family Tree	Value
------------	-------------	-------

Initial Value

Queue Token Index: m

LowerBound: 1

UpperBound: 1

Index	Lower bound	Upper bound
-------	-------------	-------------

Value: k

OK

Cancel

Apply

Validate Form

Print

Call Form for queue icon Qout

PGMT

Call Form

Icon Family Name: Qout

Prototype Name: funnyQueue1

Icon Family Tree

add

del

Index	Lower bound	Upper bound
-------	-------------	-------------

Actual Type Arguments

Formal Type	Base Type
-------------	-----------

Actual Mode Arguments

Formal Height	Actual Height	Formal Base Type	Actual Base Type
height	2	base_type	char

Actual GIP Bindings

Formal GIP	Family Tree	Value
how_broad	<input type="checkbox"/> Fix Family Tree	breadth

Initial Value

Queue Token Index:

LowerBound: 1

UpperBound: 0

Index	Lower bound	Upper bound

Value:

OK

Cancel

Apply

Validate Form

Print

Note: *breadth* is a graph formal GIP so the Family Tree table for *how_broad* is ignored and cannot be edited.

Icon Arc Forms

Icon Arc Form for transition icon wildOne

PGMT

Icon Arc Form

Icon Family Name: wildOne

Input Ports

Input Port Family Na...	From Icon Family Na...	From Port Family Name	Open Arc Form
feedBackIn	Qfeedback	OUTPUT	<input type="checkbox"/> open ...
passIn	Qin	OUTPUT	<input type="checkbox"/> open ...
passIn	Qin	OUTPUT	<input type="checkbox"/> open ...

Output Ports

Output Port Family N...	To Icon Family Name	To Port Family Name	Open Arc Form
passOut	Qout	INPUT	<input type="checkbox"/> open ...
feedBackOut	Qfeedback	INPUT	<input type="checkbox"/> open ...

OKPrint

Icon Arc Form for queue icon Qin

PGMT

Icon Arc Form

Icon Family Name: Qin

Input Ports

Input Port Family ...	From Icon Family...	From Port Family...	Open Arc Form
-----------------------	---------------------	---------------------	---------------

Output Ports

Output Port Famil...	To Icon Family N...	To Port Family N...	Open Arc Form
OUTPUT	wildOne	passIn	<input type="checkbox"/> open ...
OUTPUT	wildOne	passIn	<input type="checkbox"/> open ...

OKPrint

Icon Arc Form for queue icon Qfeedback

PGMT

Icon Arc Form

Icon Family Name: Qfeedback

Input Ports

Input Port Family ...	From Icon Family...	From Port Family...	Open Arc Form
INPUT	wildOne	feedBackOut	<input type="checkbox"/> open ...

Output Ports

Output Port Famil...	To Icon Family N...	To Port Family N...	Open Arc Form
OUTPUT	wildOne	feedBackIn	<input type="checkbox"/> open ...

OK

Print

Icon Arc Form for queue icon Qout

PGMT

Icon Arc Form

Icon Family Name: Qout

Input Ports

Input Port Family ...	From Icon Family...	From Port Family...	Open Arc Form
INPUT	wildOne	passOut	<input type="checkbox"/> open ...

Output Ports

Output Port Famil...	To Icon Family N...	To Port Family N...	Open Arc Form
----------------------	---------------------	---------------------	---------------

OK

Print

Arc Forms

Arc Form Qfeedback -> wildOne

PGMT

Arc Form

Nested Loop

add

del

Index	Lower bound	Upper bound
k	1	breadth

Connect

	Family Name	Index	Assigned Value
From Icon:	Qfeedback	k	k
Output Port of the From Icon:	OUTPUT		
To Icon:	wildOne	k	k
Input Port of the To Icon:	feedBackIn		

When: true

OK

Cancel

Apply

Validate Form

Print

Arc Form wildOne -> Qfeedback

PGMT

Arc Form

Nested Loop

add

del

Index	Lower bound	Upper bound
k	1	breadth

Connect

	Family Name	Index	Assigned Value
From Icon:	wildOne	k	k
Output Port of the From Icon:	feedBackOut		
To Icon:	Qfeedback	k	k
Input Port of the To Icon:	INPUT		

When: true

OK

Cancel

Apply

Validate Form

Print

Arc Form Qin -> wildOne (1)

PGMT

Arc Form

Nested Loop

add

del

Index	Lower bound	Upper bound
i	0	length - 1
j	0	2 * width - 1
k	0	breadth - 1

Connect

	Family Name	Index	Assigned Value
From Icon:	Qin	i	i
		j	j
		k	k
Output Port of the From Icon:	OUTPUT		
To Icon:	wildOne	k	k + 1
Input Port of the To Icon:	passIn	r	i
		s	j

When:

(j * 2) / 2 <= 0

OK

Cancel

Apply

Validate Form

Print

Arc Form Qin -> wildOne (2)

PGMT

Arc Form

Nested Loop

add

del

Index	Lower bound	Upper bound
i	0	length - 1
j	0	2 * width - 1
k	0	breadth - 1

Connect

	Family Name	Index	Assigned Value
From Icon:	Qin	i	i
		j	j
		k	k
Output Port of the From Icon:	OUTPUT		
To Icon:	wildOne	k	k + 1
Input Port of the To Icon:	passIn	r	length - 1 - i
		s	j

When: (j * 2) / 2 > 0

OK

Cancel

Apply

Validate Form

Print

Arc Form wildOne -> Qout

PGMT

Arc Form

Nested Loop

add

del

Index	Lower bound	Upper bound
k	1	breadth

Connect

	Family Name	Index	Assigned Value
From Icon:	wild One	k	k
Output Port of the From Icon:	passOut		
To Icon:	Qout		
Input Port of the To Icon:	INPUT	i	k - 1

When:

true

OK

Cancel

Apply

Validate Form

Print

System supplied prototypes

Transition Prototype Forms

Pack Prototype Form:

PGMT

Prototype

Prototype Name: Pack Entity Type: transition

Formal Type Arguments

Formal Mode Arguments

Token Height (Formal Name)	Base Type (Formal Name)
height	base_type

Formal Graph Instantiation Parameters (GIPs)

Name	Height (Number)	Base Type (Actual)

Input Ports

Family Name	Category	Token Ht	Base Type	Family Tree
INPUT	transition	height	base_type	
READAMOUNT	transition	0	unsigned int	
READOFFSET	transition	0	unsigned int	
CONSUMEAM...	transition	0	unsigned int	
CONSUMEOF...	transition	0	unsigned int	

Output Ports

Family Name	Category	Token Ht	Base Type	Family Tree
OUTPUT	transition	height+1	base_type	

OK
Print

Note that

- the form is read-only so there are no *add* or *delete* buttons in any table,

- the Formal Mode argument shows a "height" and "base_type"; these are used in two ports: INPUT and OUTPUT,
- the Family Tree of every port of the Pack is empty so the cells in the Family Tree columns are blank,
- the button to open the Body is missing, because the Pack is a special transition and has a non-standard Transition Statement.

Unpack Prototype Form:

PGMT

Prototype

Prototype Name: **Unpack** Entity Type: **transition**

Formal Type Arguments

Formal Name

Formal Mode Arguments

Token Height (Formal Name)	Base Type (Formal Name)
height	base_type

Formal Graph Instantiation Parameters (GIPs)

Name	Height (Number)	Base Type (Actual)

Input Ports

Family Name	Category	Token Ht	Base Type	Family Tree
INPUT	transition	height+1	base_type	

Output Ports

Family Name	Category	Token Ht	Base Type	Family Tree
OUTPUT	transition	height	base_type	
PRODUCE	transition	0	unsigned int	

OK **Print**

OK	Print
----	-------

Note that

- the form is read-only so there are no *add* or *delete* buttons in any table,
- the Formal Mode argument shows a "height" and "base_type"; these are used in two ports: INPUT and OUTPUT,
- the Family Tree of every port of the Pack is empty so the cells in the Family Tree columns are blank,
- the button to open the Body is missing, because the Unpack is a special transition and has a non-standard Transition Statement.

Standard Place Prototype Forms

In the following forms of standard Queue and standard Graph Variable, the Family Tree buttons are missing, because the Family Trees are empty to indicate that the Port Family Height is 0 and that the ports are single ports.

Standard Queue Prototype Form:

PGMT				
Place Prototype				
Prototype Name: Queue		Entity Type: queue		
Formal Mode Argument				
Token Height (Formal Name)		Base Type (Formal Name)		
height		base_type		
Formal Graph Instantiation Parameters (GIPs)				
Name	Height (Number)	Base Type (Actual)		
Input Port				
Family Name	Category	Token Ht	Base Type	Family Tree
INPUT	place	height	base_type	
Output Port				
Family Name	Category	Token Ht	Base Type	Family Tree
OUTPUT	place	height	base_type	
OK Print				

Standard Graph Variable Prototype Form:

PGMT

Place Prototype

Prototype Name: GVar Entity Type: gvar

Formal Mode Argument

Token Height (Formal Name)	Base Type (Formal Name)
height	base_type

Formal Graph Instantiation Parameters (GIPs)

Name	Height (Number)	Base Type (Actual)
------	-----------------	--------------------

Input Port

Family Name	Category	Token Ht	Base Type	Family Tree
INPUT	place	height	base_type	

Output Port

Family Name	Category	Token Ht	Base Type	Family Tree
OUTPUT	place	height	base_type	

OK Print

References

1. [PGM specification](#)
2. Dick Stevens: Design Notes for GUI Forms
3. Dick Stevens: Specification for the Translator Output
4. Roger Hillson: Mtool/PGMT Integration